Atty Docket No. 02998.P013 *Patent*

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

In re Application of:

Wolf-Dietrich Weber

Application No.: 09/802,405

Filed: March 1, 2001

For: COMMUNICATIONS SYSTEM AND METHOD WITH NON-BLOCKING SHARED INTERFACE

Examiner: Siddiqi, Mohammad A.

Art Group: 2154

Confirmation No.: 5453

Mail Stop Appeal Brief- Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF IN SUPPORT OF APPELLANT'S APPEAL

Applicants (hereinafter "Appellant") hereby submits this Brief as an appeal to the Board of Patent Appeals and Interferences (hereinafter "Board") from the decision of the Examiner of Group 2154, dated March 11, 2005, which finally rejected Claims 1-36 in the above-identified application. This Appeal Brief is hereby submitted pursuant to 37 C.F.R. § 41.37(a).

An appeal conference is desired.

FIRST CLASS CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail with sufficient postage in an envelope addressed to Mail Stop Appeal Brief- Patents, Commissioner for Patents, PO Box 1450, Alexandria, Virginia 22313-1450 on November 4, 2005.
Date of Deposit

Krista Matheison
Name of Person Mailing Correspondence

Krista Matheison
Signature

Nov. 4, 2005
Date

11/08/2005 CNGUYEN 00000069 09802405

02 FC:1402 500.00 OP

TABLE OF CONTENTS

I. REAL PARTY IN INTEREST

The real party in interest is the assignee of the full interest in the invention, Sonics, Inc., 2440 W. El Camino Real, Suite 620, Mountain View, California 94040.

II. RELATED APPEALS AND INTERFERENCES

To the best of Appellant's knowledge, there are no appeals or interferences related to the present appeal that will directly affect, be directly affected by, or have a bearing on the Board's decision in the instant appeal.

III. STATUS OF THE CLAIMS

Claims 1-36 are pending in the application and were finally rejected in an Office Action mailed March 11, 2005. Claims 11-3, 17-19, 28-30, 35 and 36 stand objected to. Claims 1-36 are the subject of this appeal. A copy of Claims 1-36, as they stand on appeal are set forth in an Appendix of Claims. Appellant notes that these claims are presented as amended in Appellant's Response to Final Office Action, filed July 11, 2005. Claims 1-36 are being appealed.

IV. STATUS OF AMENDMENTS

An amendment was filed on July 11, 2005, subsequent to the Final Office Action mailed March 11, 2005. The Examiner confirmed the final rejection of these claims in an Advisory Action mailed August 08, 2005 (hereinafter "Advisory Action"). A copy of all claims on appeal is attached hereto as an Appendix of Claims.

Appellant respectfully traverses each of these grounds of rejection.

V. SUMMARY OF CLAIMED SUBJECT MATTER

According to one embodiment, a method for communicating data between functional blocks in a computing device in a multi-threading system is described

in independent claim 1. The method establishes a thread identifier for each independent data stream between an initiator functional block and a target functional block. (Specification at page 7, ¶ 0012; page 8 ¶ 0013; page 32 ¶ 0077.) A plurality of independent data streams exist between a first initiator functional block and a first target functional block. (Specification at page 34 ¶ 0084; page 39 ¶ 0095-0096; figure 4 and 12.) The method further includes a condition wherein, if the first target functional block is unable to accept a data transfer from the first initiator functional block, the first target functional block issues a busy signal identified by the thread identifier to the first initiator. (Specification at page 19 ¶ 0052; page 24 ¶ 0059; page 25 ¶ 0062; page 33 ¶ 0079-0080.) The method further states that the first initiator functional block withholds the issuing of data transfers associated with the thread identifier in response to the issued busy signal. However, the first initiator may issue data transfers that are not associated with the thread identifier identified by the issued busy signal to the first target. (Specification at page 23 ¶ 0058; page 24 ¶ 0059-0060; page 25 ¶ 0062.) Lastly, the method includes the mapping of a data flow from the initiator functional block to the target functional block to a thread indicated by the thread identifier to meet a service guarantee on a per thread identifier basis (Specification at page 21 ¶ 0054;page 37 ¶ 0090-0092; page 40 ¶ 0098-0099; page 43 ¶ 0108-0110; figure 15a, 15b, 17, 19 and 21.) Thus, multiple data streams can exist at any given time between the first initiator and the first target. Each data stream has its own thread identifier. Even though the first target indicates its busy on a first thread identifier, the first initiator still issues

data transfers to that first target on different data streams with other thread identifiers.

VI. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

I. Claims 1-10, 14-16, 20-27 and 31-34 stand rejected under 35 U.S.C. §103(a) as being obvious over U.S. Patent Number 5,274,783 (hereinafter "House") in view of U.S. Patent Number 6,002,692 (hereinafter "Wills"). For purposes of this appeal, claims 1-36 stand and fall together as Group I.

VII. ARGUMENT

A. CLAIM GROUP I: APPLICANT'S TRAVERSE THE EXAMINER'S CONCLUSIONS ON THE MEANING CONVEYED WITH THE TERMS "THREADS", "THREAD ID'S" AND "DATA STREAM AS WOULD BE UNDERSTOOD BY ONE SKILLED IN THE ART."

Overall, a reference used by the office action named House discloses a single threaded system that provides a protocol and format for communications within a system. In a single threaded system, as one data stream thread is completed between a target and an initiator, then in a sequential fashion a second data stream may be initiated between those same two devices. In multi-threading systems, a single source device/application may have two or more threads of data streams in progress at the same time between itself and a target. In multi-threading systems, when an initiator receives a response from a particular target, the initiator must know more than just the source ID of target to figure out what data stream this response belongs to. Multiple data streams may

be occurring between the initiator and target at the same time. The initiator should also know the thread id of the data stream that the response from the target came from. The office action tries to stretch the interpretation of the terms in applicant's claims so that the single threaded system disclosed by House can be used to satisfy clearly stated limitations in applicant's claims.

Claim 1 in its context and entirety states:

> A method for communicating data between functional blocks in a computing device, comprising:
> establishing a thread identifier for each independent data stream between an initiator functional block and a target functional block, wherein a plurality of independent data streams exist between the initiator functional block and the target functional block,
> if the target functional block is unable to accept a data transfer from the initiator functional block, the target functional block issuing a busy signal identified by the thread identifier;
> the initiator functional block withholding issuance of data transfers associated with the thread identifier in response to the issued busy signal, wherein data transfers not associated with the thread identifier identified by the issued busy signal may be issued; and
> mapping a data flow from the initiator functional block to the target functional block to a thread indicated by the thread identifier to meet a service guarantee on a per thread identifier basis.

Appellant submits that the Examiner has improperly construed definitions of the terms "threads", "thread IDs" and "data stream." The Examiner stated:

> Examiner notes that it is the Applicant's claims [words?] as stated in the Applicant's claims that are being rejected with prior art. For example, the "data streams" of claim 1 is interpreted as a 'signals'. Signals being transmitted are information. They can be spoken words such as telephone conversation, music, or even computer data. 'thread identifier' is interpreted as 'connection identifier' or unique identifier, all of these terms are well known in the inter-process communication art.

(Office Action dated 3-11-05, page 8)

Appellant traverses the Examiner's stated conclusions on the meaning of the terms "threads" and "thread Ids" with evidence from various dictionaries and articles by people skilled in the art concerning what the above terms generally convey. Appellant submits definitions of the term "thread" from the following four technical websites and one standards body: 1) Geek.com; 2) Whatis.com; 3) Webopedia.com; 4) Techweb.com; and 5) the Open Core Protocol Specification version 2.0 published by the OCP-IP standards organization. Appellant submits articles on multi-threading from: 1) Sun Microsystems' article on multi-threading that discusses threads in JAVA software environment; 2) a Computer Programming Thread FAQ; and 3) again the Open Core Protocol Specification version 2.0 published by the OCP-IP standards organization. The entire Open Core Protocol Specification document is relevant to person's skilled in the art in an understanding of the above terms but appellant specifically directs the Examiner's attention to pages 10, 19, 20, and 46 of this large document.

The seven references discussing threads and multi-threading all differ slightly but convey a common theme. The theme is that a "thread" can be thought of as 1) a series of communication exchanges, usually involving data, in a related transaction between an initiating device to a target device and/or 2) a part of a single program that runs independently and/or simultaneously along with other threads from that program to accomplish a specific task. Either way, a single source device/application may have two or more threads of data streams in progress at the same time between itself and a target.

Appellant submits that the Examiner's definition of 'thread identifier' as being interpreted as 'connection identifier' or unique identifier is incorrect. A thread identifier identifies a thread, whereas a connection identifier and a unique identifier can encompass a much different meaning. For example, a unique identifier could be used to identify anything at all, from a vehicle identification number to a cellular phone's serial number. A connection identifier may identify a source address and a target address. However, a connection identifier may not in all cases uniquely identify the particular data stream thread occurring between that source address and target address. Hence, Appellant submitted the definitions to show that one reasonably skilled in the art would not reasonably equate a thread id associated with a data stream in a multi-threaded system with a unique identifier associated with spoken words in a telephone conversation. (See Examiner's assertion in Office Action dated 3-11-05, pages 6-8)

Similarly, appellant traverses the Examiner's stated conclusions on the meaning of "data stream" with evidence from various dictionaries by people skilled in the art concerning what the term generally conveys. Appellant submits definitions of the term "data stream" from the following technical website, standards body, and the Wills reference cited by the Examiner in the two previous office actions: 1) Techweb.com; 2) the American National Standard's Telecom glossary; and 3) the Wills reference describes a SONET STS-48 data stream at Col. 4 Line 62 to Col. 5 Line 14.

The three references discussing the term "data stream" all differ slightly but convey a common theme. The theme is that a "data stream" can be thought

of as "a sequence of encoded information in a transmission from one place to another."

Appellant submits that the Examiner's narrow definition of 'data stream' as being interpreted as 'signals' is incorrect. The Examiner's statement that, "Signals being transmitted are information. They can be spoken words such as telephone conversation, music, or even computer data" is too broad of a meaning, going far beyond the narrower definition of "a sequence of encoded information in a transmission from one place to another."

Appellant submitted the above definitions for the three terms being discussed in the Response after Final Rejection, dated 06/11/05.

B. CLAIM GROUP I: BASED ON A REASONABLE INTERPRETATION OF THE TERMS "THREADS", "THREAD ID'S" AND "DATA STREAMS", THE COMBINATION OF HOUSE AND WILLS DO NOT MAKE CLAIMS 1-10, 14-16, 20-27 AND 31-34 OBVIOUS UNDER 35 U.S.C. §103.

1. The office action fails to make an adequate rejection under 35 U.S.C. §103 pointing out how the references make applicant's claims obvious.

The office action rejects claims 1-10, 14-16, 20-27, and 31-34 under 35 U.S.C. § 103(a) as being unpatentable over House in view of Wills. The Examiner states:

> Applicant's arguments filed 10/21/2004 have been fully considered but they are not persuasive.
>
> In the remarks applicants argued:
>
> A. House does not disclose the existence of multiple independent data.
>
> B. House does not disclose establishing the thread identifier for each independent data stream.
>
> C. House does not disclose actually issuing data transfers not associated with thread identifier.
>
> . . .
>
> Examiner notes that while specific references were made to the prior art, it is actually also the prior arts in its entirety and the combination of the prior arts in its entirety that is being referred to.
>
> (Office Action dated 3-11-05, pages 6-8)

However, applicant respectfully asserts that claim 1, as amended, is not obvious under 35 U.S.C. § 103(a) in view of the combination of House with Wills.

The law requires:

> To establish a prima facie case of obviousness, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. In re Vaeck, 947 F.2d 488, 20 USPQ2d 1438 (Fed. Cir. 1991). (Manual of Patent Examining Procedure ¶ 2143).

Further, the law also requires:

> To establish *prima facie* obviousness of a claimed invention, all the claim limitations must be taught or suggested by proposed

combination of the prior art. *In re Royka*, 490 F.2d 981, 180 USPQ 580 (CCPA 1974). (Manual of Patent Examining Procedure (MPEP) ¶ 2143.03).

The first point in patent law requires the combination of the references themselves to disclose each and every limitation in applicant's claims to render those claims obvious under section 103. Further, the Examiner is required to particularly point out where these limitations are disclosed in the prior art document. Thus, if the office action makes a specific reference to where a claim limitation is disclosed in prior art document and in fact that limitation is not disclosed in that reference, then the office action is not satisfying the requirements of section 103. Also, a limitation maybe inherently disclosed under 103 if the Examiner makes a proper evidentiary finding. The PTO may establish that one of ordinary skill in the art would have been motivated to combine the references with articulated findings of fact regarding: 1) the level of skill in the art; 2) the relationship between the fields of the cited art; and 3) the particular features of the prior art references that would motivate one of ordinary skill in applicant's particular art to select elements disclosed in references from a wholly different field. In re Dembiczak, 175 F.3d 994, 996 (Fed. Cir. 1999). However, the office action's statement "Examiner notes that while specific references were made to the prior art, it is actually also the prior arts in its entirety and the combination of the prior arts in its entirety that is being referred to" does not satisfy that evidentiary requirement. The above statement does not either 1) makes a specific reference to where the limitation is found in a specific reference document or 2) make a proper evidentiary finding of fact.

2. House discloses a single threaded communication system that does not disclose the existence of multiple independent data streams between an initiating device and a target device.

The law requires:

> To establish *prima facie* obviousness of a claimed invention, all the claim limitations must be taught or suggested by proposed combination of the prior art. *In re Royka*, 490 F.2d 981, 180 USPQ 580 (CCPA 1974). (Manual of Patent Examining Procedure (MPEP) ¶ 2143.03).

Appellant submits that House fails to disclose the existence of multiple independent data streams occurring concurrently between two functional blocks. In contrast, House merely discloses a bus extender which contains transfer and logic circuitry that implement a single data stream between devices. House discloses:

> In particular, the transfer and logic circuitry (i) receives first connection-control signals from one of the buses, which signals have fields of data designating the extender as the addressee and designating the source of the signals; (ii) identifies the ultimate target for the inter-bus communication based on data contained in the first connection-control signals, or, depending on the direction of the communication, stored in a latch within the extender itself; (iii) generates second connection-control signals including fields of data designating the extender as the source of the communication and the ultimate target; and (iv) provides these latter signals to the appropriate transceiver for transmission over the other bus.

(House, col. 2, lines 40-53) (emphasis added)

House discloses a method for creating a single data stream between devices connected to an auxiliary bus and a main bus. Each communication originating from a device connected to the auxiliary bus to

the main bus is processed by the transfer and logic circuitry as detailed above. House discloses how to set up a single data stream between two devices; however, House is silent about multiple data streams can exist between on device and another. House does not disclose any details like assigning multiple thread ids because multiple independent data stream may exist. House is silent about these details because House does not disclose or suggest the existence of multiple independent data streams between one device and another. House merely discloses that all data originating from one device and having the same destination is processed in the same fashion. Therefore, House does not disclose "a plurality of independent data streams" between a device on the auxiliary bus and the main bus.

3. House does not disclose establishing a thread identifier for each independent data stream between two functional blocks.

Appellant submits that House fails to disclose establishing a thread identifier for each independent data stream between two functional blocks. In contrast, House merely discloses assigning a single identification code to a device connected to the main bus. House discloses:

> [T]he bus extender takes advantage of dual-tier, hierarchal addressing used in the SCSI standards to direct messages to the designated devices on the other bus. In the addressing scheme employed in the invention, each device connected to either the main or auxiliary bus is identified by a unique identification code ("ID"). In addition, each device ID is associated with an auxiliary identification or address-descriptor, which in the SCSI standards is referred to as a LUN or logical unit number.

. . .
For purposes of communication, the bus extender has an ID on both the main and auxiliary buses.
. . .
In order to pass messages received over the main bus from a host computer, i.e., during SELECTION, the extender first converts the LUN field data of the connection-control signals received over the main bus to the ID of the target on the auxiliary bus, and supplies the extender's own auxiliary-bus ID as the initiator ID in the auxiliary-bus connection-control signals.

(House, col. 2, lines 54 through col. 3 line 9) (emphasis added)

Accordingly, House discloses that a device has a single ID associated with it. House does not disclose "establishing a thread identifier for each independent data stream between an initiator functional block and a target functional block, wherein a plurality of independent data streams exist between the initiator functional block and the target functional block."

4. House does not disclose the issuance of data transfers not associated with the busy thread identifier from the initiator while withholding issuance of data transfers associated with the busy thread identifier.

The Examiner continuously cites House at Col. 9 Lines 10-36 as disclosing "the initiator functional block withholding issuance of data transfers associated with the thread identifier in response to the issued busy signal, wherein data transfers not associated with the thread identifier identified by the issued busy signal may be issued." Specifically, the Examiner states:

> if the target functional block is unable to accept a data transfer from the initiator functional block (col. 9, lines 10-25), the target functional block issuing a busy signal identified by the thread identifier (col. 9, lines 25-36);

the initiator functional block withholding issuance of data transfers associated with the thread identifier in response to the issued busy signal (col. 9, lines 20-36), wherein data transfers not associated with the thread identifier identified by the issued busy signal may be issued (col. 9, lines 25- 36)
(Office Action dated 3-11-05, page 7)

Applicants assert the House does not disclose that the same initiator withholds issuance of data transfers associated with the busy thread identifier and allows the issuance of data transfers from a data stream that are not associated with the busy thread ID. In contrast, House merely discloses arbitration between two devices to obtain control of a bus and the subsequent establishment of a communication link between the initiating host and a target. The control logic allows the establishment of a communication link between the first initiating device and a first target while issuing a busy signal and blocks transactions from any additional initiating devices.

Accordingly, House discloses arbitration between two devices, a host computer 14 and an extender 30 to obtain control of a bus:

> To transfer messages, the host computer 14 attempts to gain control of the main bus 26 during what is called the ARBITRATION phase by asserting the BUSY line ("bsy") at "a" in part 7A of the drawing, and asserting the host computer's own ID on the data lines ("dbn") at "b." (In SCSI buses, there are, e.g., eight data lines, one corresponding to each of the ID's (i.e., ID.sub.-- 0-ID.sub.-- 7) that can be assigned to devices on the bus. Thus, for example, to assert ID.sub.-- 6, the sixth data line is driven HIGH.)
>
> If at the time the host computer 14 is attempting to control the main bus 26, any other device or devices are likewise attempting to do so, the bus is deemed to be in contention. In that case, according to the SCSI standards, the contending device with the highest ID is given priority. Thus, for example, if the extender 30 were also attempting to control the main bus

> 26, the control logic 50 would assert BUSY and the extender's ID, i.e., ID.sub.-- 0, on the data lines. Since, the computer's ID.sub.-- 6 is higher than the extender's ID.sub.-- 0, the extender 30 would fall off the main bus 26, and the host computer 14 would gain control of the main bus 26 by asserting the SELECT line ("sel"). If there are no contenders for the bus, then the host computer 14 simply can assert "sel," as shown at "c." This finishes the main-bus ARBITRATION phase.
> (House Col. 8 Ln. 62 to Col. 9 Ln. 9)

Next, House discloses that the initiating device, host computer 14, establishes a communication link with a target, one of the auxiliary-bus peripheral devices 24. A busy signal is asserted on the main bus during the establishment of this communication link. House discloses:

> Now, the host computer 14 attempts to establish a communication link with a target on the main bus 26 within what is called the SELECTION phase by sending a first connection control signal over the data lines ("dbn") as shown at "d," which signal gives both the host computer's own ID as the initiator and the target's ID on the main bus 26. Consequently, two of the data lines are asserted--the two corresponding to the host computer and the target. In addition, the correct parity for the asserted data bits, i.e., in this case, a HIGH value, is maintained on the data parity line ("dbp"). In other words, dbp is asserted at "e.". Moreover, another signal line, the I/O control line ("i/o"), is deasserted to indicate SELECTION. (Assertion of the i/o line indicates RESELECTION.) Afterwards, the initiator also deasserts BUSY at "f."
>
> In order to illustrate the invention, we will assume that the target ID asserted during SELECT is that of the bus extender 30, e.g., ID.sub.-- 0, which means that the host computer 14 is attempting to communicate with one of the auxiliary-bus peripheral devices 24. Accordingly, during SELECT, the control logic 50 of the extender 30 identifies the target as ID.sub.-- 0, and verifies that SELECT is asserted and that BUSY is deasserted. In addition, the control logic 50 verifies that the parity is correct, and that there are two, and only two, bits asserted on the dbn lines.

> Once the extender 30 has confirmed that it is the target, the extender accepts SELECTION by asserting BUSY on the main bus 26, as shown at point "g" in FIG. 5. In response to the acceptance, the host computer 14 de-asserts SELECT at "h." (House Col. 9 Lines. 10-41)

Appellant asserts that House, in the above section cited by the Examiner, does not disclose that the same initiator withholds issuance of data transfers associated with the busy thread identifier and allows the issuance of data transfers from a data stream that are not associated with the busy thread ID. To reiterate, House merely discloses arbitration between two devices to obtain control of a bus. House then discloses the subsequent establishment of a communication link between the initiating host and a target while issuing a busy signal and blocks transactions from any additional initiating devices.

On page 2 of the advisory action dated 08/08/05, the Examiner states:

> Regarding House not teaching data transfer between target and initiator. Applicant's attention is drawn to elements of figs 5 and 6 of [the] House reference in which the busy signal asserted, while other data lines transferring data/messages (col. 9, lines 26-67).

Appellant submits that House teaches data transfer in general. As stated above, House merely teaches arbitration between two devices to obtain control of a bus. House then discloses the subsequent establishment of a communication link between the initiating host and a target. Such communication is that of single threaded data transfers and not multi-threaded data transfers. Therefore, the data transfers taught by House are insufficient.

5. Wills does not disclose meeting service guarantees on a per thread identifier basis.

Appellant submits that Wills does not disclose meeting service guarantees on a per thread identifier basis. In contrast, Wills merely discloses:

> The converter also splits the traffic into multiple priorities so as to assure quality of service (QoS) for timing critical traffic.
> (Wills, col. 5, lines 19-21)

Wills merely discloses a method for ensuring quality of service by solely prioritizing traffic. Wills discloses splitting traffic to meet a quality of service guarantee, but does not differentiate between two data streams originating from the same functional block yet belonging to two different thread identifiers. Hence, Wills does not teach, suggest or disclose meeting service guarantees on a per thread basis.

C. CLAIM GROUP I: NO MOTIVATION EXISTS TO COMBINE HOUSE WITH WILLS TO ACHIEVE THE ELEMENTS OF APPELANTS' INVENTION.

Appellant submits that the Examiner has provided inadequate motivation to properly combine the references, House and Wills, under 35 U.S.C. § 103. Patent law requires that the evidence for the motivation to combine references under 35 U.S.C. § 103 must come from either 1) within the references themselves or 2) make articulated findings of fact regarding: A) the level of skill in the art; B) the relationship between the fields of the cited art; and C) the particular features of the prior art references that would motivate one of ordinary skill in applicant's particular art to select elements disclosed in references. See In re Lee, 277 F.3d 1338, 1344 (Fed. Cir. 2002), In re Thrift, 298 F.3d 1357, 1361

(Fed. Cir. 2002), In re Dembiczak, 175 F.3d 994 (Fed. Cir. 1999), and the Manual of Patent Examining Procedure section 2143. The Examiner merely states that:

> House does not specifically discloses to meet a service guarantee on a per thread identifier basis. However, Will discloses to meet a service guarantee on a per thread identifier basis (col. 5, lines 19-34).
>
> Therefore, it would have been obvious to one of ordinary skill in the art at the time *of* the invention was made to combine Will with House because it would provide a guaranteed throughput.

(Office Action dated 3-11-05, pages 3-4)

Paraphrasing the above language, Reference A does not disclose limitation X but Reference B does and it's obvious to combine them to achieve a result, presumed by law, found in applicant's patent application. The Examiner cites to no hints or suggestions in either reference that explicitly or implicitly suggests the combination of these two references. The Examiner fails to make an articulated finding of fact. Therefore, on this basis alone, applicant respectfully submits that a presumed impressible use of hindsight has occurred and the obviousness rejection of claims 1-10, 14-16, 20-27 and 31-34 has been overcome.

D. APPELLANT CORRECTLY ESTABLISHES A CORRELATION BETWEEN THE CLAIMED INVENTION AND DEFINITIONS FOR 'THREADS', 'THREAD ID'S' AND 'DATA STREAM.'

On page 2 of the advisory action dated 08/08/05, the Examiner states:

> Applicant failed to establish correlation between claimed invention and definitions. Applicant fails to point out where the threading model clearly and concisely described/enabled/implemented/used in the specification as it is defined in cited definitions after the final rejection.

Appellant submits that there is a proper correlation between the claimed invention and the definitions of "threads", "thread IDs" and "data streams" as defined in the response dated 06/11/05 and restated in the current appeal brief. Applicants assert that a nexus properly exists between the definitions above and the claims. On page 8 of the office action dated 03/11/05, the Examiner offers definitions for the terms "threads", "thread IDs" and "data stream" that were in contrast with what appellant generally believes one reasonably skilled in the art would understand that these terms convey. Appellant believes these definitions to be incorrect and submitted what appellant believes generally to be the correct definitions used by others skilled in the art, by citing to numerous, widely used technical references. Hence the correlation between the cited definitions and the claims are relevant since the terms "threads", "thread IDs" and "data streams" are used directly in the claims. For example, claim 1 uses these three terms 9 times. Thus, a nexus properly exists between the submitted definitions for the terms in dispute and the current claims in this office action.

Appellant submits that terms used in the claims are given its ordinary meaning as known to others skilled in the art, unless a contrary definition is described in the specification. Appellant's specification does not created special definitions for the three terms above. Instead the appellant relied on the ordinary meaning of these terms as known to other skilled in the art. The citations to the external references only attempt to show a general meaning of what the ordinary meaning of these terms are known by other skilled in the art.

E. APPELLANT'S ARGUMENTS PROPERLY COMPLY WITH C.F.R. §1.111(C).

On page 2 of the advisory action dated 08/08/05, the Examiner states:

> Applicant's arguments do not comply with 37 C.F.R. §1.111(c) because they do not clearly point out the patentable novelty which he or she thinks the claims present in view of the state of the art disclosed by the references cited or the objections made.

Appellant's are confused by this statement. Appellant's responses dated 10/21/04 and 06/11/05 specifically lists multiple reasons why the claims present patentable novelty over the state of the art disclosed by House and Wills. The Examiner acknowledged the relevance of the arguments when the Examiner stated "Applicant's arguments filed 10/21/2004 have been fully considered but they are not persuasive. (Office Action dated 6-11-05 page X). Moreover, the examiner then summarized the relevance of appellant arguments. These reasons have again been recited in the arguments section of this appeal brief. Specifically, the amendments address the following distinctions between the claims and the cited references:

1. House does not disclose the existence of multiple independent data streams between an initiating device and a target device. Office Action Response dated 10/21/04, page 15 and Office Action Response dated 06/11/05, page 19, last paragraph.

2. House does not disclose establishing the thread identifier for each independent data stream. Office Action Response dated 10/21/04,

page 16 and Office Action Response dated 06/11/05, page 19, last paragraph.

3. House does not disclose the issuance of data transfers not associated with the busy thread identifier from the initiator while withholding issuance of data transfers associated with the busy thread identifier. Office Action Response dated 10/21/04, page 16-18 and Office Action Response dated 06/11/05, page 16-19.

4. Wills does not disclose meeting service guarantees on a per thread identifier basis. Office Action Response dated 10/21/04, page 18 and Office Action Response dated 06/11/05, page 19, last paragraph.

Thus, Applicant's arguments clearly comply with 37 C.F.R. §1.111(c) by clearly point out the patentable novelty which he or she thinks the claims present in view of the state of the art disclosed by the references cited.

F. PRAYER FOR RELIEF.

With all due respect, the prosecution of this case on its merits has been exhausted. Appellant respectfully request that the Appeal Board use its authority to allow these claims. In the previous responses, the applicant responded to the same issues as well as unwarranted 35 USC 112 rejections. In other responses, the applicant has over came a different set of cited prior art references. The claims have not changed since the last response to an office action and no

meaningful examination is occurring in this case. Appellant respectfully submits for all of the reasons discussed above claims 1-36 should be allowed.

IX. CONCLUSION

For the reasons stated above, the rejection of claims 1-10, 14-16, 20-27 and 31-34 under 35 U.S.C. § 103(a) of House over Wills should be withdrawn. Appellant respectfully requests that the Board reverse the rejections of the claims under 35 U.S.C. §103(a), and since there are no remaining grounds of rejection to be overcome, direct the Examiner to enter a Notice of Allowance for Claims 1-36.

Fee for Filing a Brief in Support of Appeal

Enclosed is a check in the amount of $620.00 to cover the fee for filing a brief in support of an appeal as required under 37 C.F.R. § 1.17(c) and 41.20(b)(2) as well as the fee for a one month extension. Appellant believes that a one month extension is all that is necessary under 37 C.F.R. § 41.39(b). For appellant believes that the Examiner's answer in their advisory action dated 08/08/2005 contains rejections designated as a new ground of rejection. Under rule 41.39(b), the time for response is set for two months from the date of the Examiner's answer. In this case, that date for reply is 10/08/05.

Deposit Account Authorization

Authorization is hereby given to charge our Deposit Account No. 02-2666 for any charges that may be due. Furthermore, if an additional extension is required over the one-month extension already included, then Appellant hereby requests such extension.

Respectfully submitted,

BLAKELY, SOKOLOFF,
TAYLOR & ZAFMAN LLP

Dated: 11-04-05

Aslam A. Jaffery
Attorney for Appellant
Registration No. 51,841

12400 Wilshire Boulevard
Seventh Floor
Los Angeles, CA 90025-1026
(303) 740-1980

IX. APPENDIX OF CLAIMS (37 C.F.R. § 41.37(c)(1)(viii))

The claims on appeal read as follows:

1. (Previously Presented) A method for communicating data between functional blocks in a computing device, comprising:

establishing a thread identifier for each independent data stream between an initiator functional block and a target functional block, wherein a plurality of independent data streams exist between the initiator functional block and the target functional block,

if the target functional block is unable to accept a data transfer from the initiator functional block, the target functional block issuing a busy signal identified by the thread identifier;

the initiator functional block withholding issuance of data transfers associated with the thread identifier in response to the issued busy signal, wherein data transfers not associated with the thread identifier identified by the issued busy signal may be issued; and

mapping a data flow from the initiator functional block to the target functional block to a thread indicated by the thread identifier to meet a service guarantee on a per thread identifier basis.

2. (Original) The method as set forth in claim 1, wherein the busy signal comprises a signal that is maintained active when the target functional block is unable to accept data transfers.

3. (Original) The method as set forth in claim 1, wherein the busy signal comprises a credit signal used to communicate a number of credits that indicate how many data transfers the target functional block can accept.

4. (Original) The method as set forth in claim 3, further comprising decrementing the number of credits for each active data transfer and incrementing the number of credits upon freeing up of resources for further data transfers.

5. (Original) The method as set forth in claim 3, wherein the credit signal is generated by maintaining the signal in an active state for a number of clock cycles corresponding to the number of credits.

6. (Previously Presented) The method as set forth in claim 3, wherein the credit signal comprises a multi-bit coded signal indicative of the number of credits.

7. (Original) The method as set forth in claim 1, further comprising determining service guarantees for at least one transaction stream between initiator functional blocks and the target functional blocks.

8. (Original) The method as set forth in claim 1, further comprising the initiator functional block stopping to send data transfers so that the target functional block receives no more than a determined number of data transfers after issuance of the busy signal.

9. (Original) The method as set forth in claim 1, wherein the target functional block issues a busy signal no more than a determined number of clock cycles after the target functional block determines that it has insufficient buffer space to receive data transfers from an initiator functional block.

10. (Original) The method as set forth in claim 8, further comprising the target device buffering the data transfers received after issuance of the busy signal until resources become available to service the buffered data transfers.

11. (Original) The method as set forth in claim 7, wherein determining service guarantees comprises:

mapping the transaction stream to data channels of components between an initiator device and target device;

converting performance guarantees of selected data channels of the mapped transaction stream such that the guarantees of the data channels are aligned to be uniform in units; and

aggregating the guarantees of the data channels for the transaction stream.

12. (Original) The method as set forth in claim 11, wherein aggregating comprises a function selected from the group consisting of summing the guarantees of the data channels of the transaction stream, selecting the maximum guarantees of the data channels of the transaction stream, and selecting the minimum guarantees of the data channels of the transaction stream.

13. (Original) The method as set forth in claim 11, wherein the guarantees are selected from the group consisting of quality of service guarantees, performance guarantees, bandwidth guarantees, latency guarantees, maximum outstanding request guarantees and maximum variance in service latency guarantees.

14. (Previously Presented) A method for communicating data between functional blocks in a computing device, comprising:

establishing at least one thread identifier, each thread identifier associating a data transfer with a transaction stream that the data transfer between an initiator functional block and a target functional block are part of;

if the target functional block is unable to accept a data transfer from the initiator functional block, the target functional block issuing a busy signal identified by the thread identifier;

storing in a buffer data transfers received by the target functional block after issuance of the busy signal until resources become available to service the buffered data transfers, the amount of buffer sufficient to buffer any transfers that arrive after the busy signal is asserted, wherein an interface between the initiator functional block and target functional block does not block data transfers of other threads; and

mapping a data flow from the initiator functional block to the target functional block to a thread indicated by the thread identifier to meet a service guarantee on a per thread identifier basis.

15. (Original) The method as set forth in claim 14, wherein the target functional block issues a busy signal a determined number of clock cycles after the target functional block determines that it is unable to accept a data transfer from an initiator functional block.

16. (Original) The method as set forth in claim 14, further comprising the target functional block receiving no more than a determined number of data transfers after issuance of the busy signal.

17. (Original) The method as set forth in claim 14, further comprising determining service guarantees for at least one transaction stream between initiator functional blocks and the target functional blocks.

18. (Original) The method as set forth in claim 17, wherein determining service guarantees comprises:

mapping the transaction stream to data channels of components between an initiator device and target device;

converting performance guarantees of selected data channels of the mapped transaction stream such that the guarantees of the data channels are aligned to be uniform in units; and

aggregating the guarantees of the data channels for the transaction stream.

19. (Original) The method as set forth in claim 18, wherein aggregating comprises a function selected from the group consisting of summing the guarantees of the data channels of the transaction stream, selecting the maximum guarantees of the data channels of the transaction stream, and selecting the minimum guarantees of the data channels of the transaction stream.

20. (Previously Presented) A communication apparatus, comprising:

at least two functional blocks, wherein an initiator functional block communicates with a target functional block by establishing a connection;

a bus coupled to each of the functional blocks and configured to carry a plurality of signals, wherein the plurality of signals comprises a thread identifier

configured to associate a data transfer with a transaction stream between the initiator functional block and target functional block, and a credit signal identified by the thread identifier, the credit signal issued by the target functional block to indicate how many data transfers the target functional block can accept, wherein the initiator functional block associated withholds issuance of data transfers associated with the thread identifier if the credit signal indicates that the target functional block can accept no data transfers, and the bus being non-blocking, via the use of credit signals, to enable a determination of service guarantees for transaction streams between initiator functional blocks and target functional blocks.

21. (Original) The apparatus as set forth in claim 20, wherein the busy signal comprises a signal that is maintained active when the target functional block is unable to accept data transfers.

22. (Original) The apparatus as set forth in claim 20, wherein the busy signal comprises a credit signal comprising a number of credits that indicate how many data transfers the target functional block can accept.

23. (Original) The apparatus as set forth in claim 22, wherein the number of credits is decremented for each active data transfer and incremented upon freeing up of resources for further data transfers.

24. (Original) The apparatus as set forth in claim 22, wherein the credit signal is generated by maintaining the signal in an active state for a number of clock cycles corresponding to the number of credits.

25. (Previously Presented) The apparatus as set forth in claim 22, wherein the credit signal comprises a multi-bit coded signal indicative of the number of credits.

26. (Original) The apparatus as set forth in claim 20, wherein the at least one transaction stream is non-blocking enabling determination of service guarantees for transaction streams between initiator functional blocks and target functional blocks.

27. (Original) The apparatus as set forth in claim 20, wherein the target functional block further comprises a buffer to receive data transfers issued by the initiator functional block after issuance of the busy signal by the target functional block and before receipt of the busy signal by the initiator functional block.

28. (Original) The apparatus as set forth in claim 27, wherein service guarantees are determined by mapping the transaction stream to data channels of components between an initiator device and target device, converting performance guarantees of selected data channels of the mapped

transaction stream such that the guarantees of the data channels are aligned to be uniform in units, and aggregating the guarantees of the data channels for the transaction stream.

29. (Original) The apparatus as set forth in claim 28, wherein aggregating comprises a function selected from the group consisting of summing the guarantees of the data channels of the transaction stream, selecting the maximum guarantees of the data channels of the transaction stream, and selecting the minimum guarantees of the data channels of the transaction stream.

30. (Original) The apparatus as set forth in claim 26, wherein the guarantees are selected from the group consisting of quality of service guarantees, performance guarantees, bandwidth guarantees, latency guarantees, maximum outstanding request guarantees and maximum variance in service latency guarantees.

31. (Previously Presented) A communication apparatus, comprising:

at least two functional blocks, wherein an initiator functional block communicates with a target functional block by establishing a connection;

a bus coupled to each of the functional blocks and configured to carry a plurality of signals, wherein the plurality of signals comprises at least one thread

identifier configured to associate a data transfer with a transaction stream that the data transfer between an initiator functional block and a target functional block are part of; wherein if the target functional block is unable to accept a data transfer from the initiator functional block, the target functional block issuing a busy signal identified by the thread identifier and buffering data transfers received after issuance of the busy signal until resources become available to service the buffered data transfers;

a buffer coupled to the target functional block, the size of the buffer sufficient to buffer any number of data transfers that arrive on the transaction stream after the busy signal is asserted; and

wherein the bus implements a mapping algorithm to map data flow of the transaction stream and aggregate service guarantees from components between the initiator functional block and the target functional block.

32. (Original) The apparatus as set forth in claim 31, wherein the target functional block issues a busy signal a determined number of clock cycles after the target functional block determines that it is unable to accept a data transfer from an initiator functional block.

33. (Original) The apparatus as set forth in claim 31, further comprising the target functional block receiving no more than a determined number of data transfers after issuance of the busy signal.

34. (Original) The apparatus as set forth in claim 31, further comprising the target functional block determining service guarantees for at least one transaction stream between initiator functional blocks and the target functional blocks.

35. (Original) The apparatus as set forth in claim 34, wherein determining service guarantees comprises:

mapping the transaction stream to data channels of components between an initiator device and target device;

selectively converting determine guarantees of data channels of components of the mapped transaction stream such that the guarantees of the data channels are aligned to be uniform in units; and

aggregating the guarantees of the data channels for the transaction stream.

36. (Original) The apparatus as set forth in claim 35, wherein aggregating comprises a function selected from the group consisting of summing the guarantees of the data channels of the transaction stream, selecting the maximum guarantees of the data channels of the transaction stream, and selecting the minimum guarantees of the data channels of the transaction stream.

X. EVIDENCE APPENDIX

This Evidence Appendix includes the following documentation that was submitted during prosecution pursuant to 37 C.F.R. § 1.31:

1) Telecom Glossary 2000, "Development Site for Proposed Revisions to American National Standard T1.523-2001", http://www.its.bldrdoc.gov/projects/devglossary/t1g-main.html, 2000 (1 page).

2) Definition of "Data Stream", http://www.its.bldrdoc.gov/projects/devglossary/_data_stream.html, June 6, 2005 (1 page).

3) TechEncyclopedia: definition of "data stream", copyright 1981-2005, http://www.techweb.com/encyclopedia/defineterm.jhtml?term+data+stream&x=32&y=4, (1 page).

4) TechEncyclopedia: definition of "streaming data", copyright 1981-2005, http:www.techweb.com/encyclopedia/defineterm.jhtml?term=streaming+data, May 20, 2005 (1 page).

5) Definition of "Thread", www,geek.com, prior to filing date of current application (1 page)

6) Definition of "Thread", search VB.com Definitions-powered by whatis.com, http://searchvb.techtarget.com/sDefinition/0,290660,sid8_gci213139,00.html, May 20, 2005 (2 pages).

7) internet.com (Webopedia), Definition of "thread", http://www.pcwebopaedia.com./TERM/t/thread.html, May 20, 2005 (2 pages).

8) TechEncyclopedia, definition of "thread", copyright 1981-2005, http://www.techweb.com/encyclopedia/defineterm.jhtml?term=thread&x=16&y=14, May 20, 2005 (1 page).

9) TechEncyclopedia: definition of "multithreading", copyright 1981-2005, http://www.techweb.com/encyclopedia/defineterm.jhtml?term=multithreading, May 20, 2005 (1 page).

10) internet.com (Webopedia), Definition of "multithreading", last modified October 24, 2002 (1 page).

11) http://www.faqs.org/faqs/threads-faq/part1/, "comp.programming.threads FAQ [last mod 97/5/24]", May 24, 1997 (last modified date), pp. 1-18 (18 pages).

12) OCP International Partnership, "Open Core Protocol Specification", Release 2.0, OCP-IP Confidential, Document Revision 1.1.1, pp. 10, 19, 20 and 46 (118 pages).

13) PAWLAN, "Creating a Threaded Slide Show Applet", Sun Microsystems, Inc., The Source, http://java.sun.com/developmer/technicalArticles/Threads/applet/, March 16, 2001, pp. 1-6 (6 pages).

XI. RELATED PROCEEDINGS APPENDIX

There are no related proceedings and therefore no documentation to be included in this Related Proceedings Appendix.

Development Site for proposed Revisions to American National Standard

T1.523-2001

This development site, while not yet fully functional, contains the baseline document T1.523-2001, and allows you to propose revisions to this standard glossary. Just click on the "Add New or Comment" button below.

revisions to American National Standard

Telecom Glossary 2000

To view the current, official version of ANS T1.523-2001, Telecom Glossary 2000, go to website,
http://www.atis.org/tg2k/.

Foreword: The communication of facts and ideas depends upon a mutual understanding of terminology. ... [Full text]

data stream

data stream: A sequence of digitally encoded signals used to represent information in transmission.

This HTML version of Telecom Glossary 2K was last generated on Wed May 8 15:36:48 MDT 2002. References can be found in the Foreword.

Send Comments on this Def

TechEncyclopedia More than 20,000 IT terms

Results found for: data stream

data stream

The continuous flow of data from one place to another.

TERMS SIMILAR TO YOUR ENTRY

Entries before data stream

- data sink
- data sizes
- data source
- DataStage
- data store

Entries after data stream

- data striping
- data structure
- data switch
- data synchronization
- data system

DEFINE ANOTHER IT TERM

Define

Or get a random definition

TechEncyclopedia More than 20,000 IT terms

Results found for: streaming data

streaming data

Data that is structured and processed in a continuous flow, such as digital audio and video. See streaming audio and streaming video.

TERMS SIMILAR TO YOUR ENTRY

Entries before streaming data

- stream
- stream cipher
- streaming
- streaming audio
- stream-oriented file

Entries after streaming data

- streaming encoder
- streaming media
- streaming radio
- streaming server
- streaming tape

DEFINE ANOTHER IT TERM

Define

Or get a random definition

WWW.Geek.com
Defintions:

Thread - Part of a program that runs independently or along with other threads to accomplish a task. To run multiple threads you must be running on an operating system such as UNIX or Windows NT/2000/XP that supports multiple threads. The performance benefit of allowing multiple threads to run at the same time is realized mainly on multi-processing systems. Different threads run on different processors, so they can run simultaneously.

searchVB.com Definitions - powered by whatis.com

BROWSE WHATIS.COM DEFINITIONS: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z # BROWSE ALL CATEGORIES

Search whatis.com for: Search - OR - Search this site: Search

thread

1) On the Internet in Usenet newsgroups and similar forums, a thread is a sequence of responses to an initial message posting. This enables you to follow or join an individual discussion in a newsgroup from among the many that may be there. A thread is usually shown graphically as an inital message and successive messages "hung off" the original message. As a newsgroup user, you contribute to a thread by specifying a "Reference" topic as part of your message.

2) In computer programming, a thread is placeholder information associated with a single use of a program that can handle multiple concurrent users. From the program's point-of-view, a thread is the information needed to serve one individual user or a particular service request. If multiple users are using the program or concurrent requests from other programs occur, a thread is created and maintained for each of them. The thread allows a program to know which user is being served as the program alternately gets re-entered on behalf of different users. (One way thread information is kept by storing it in a special data area and putting the address of that data area in a *register*. The operating system always saves the contents of the register when the program is interrupted and restores it when it gives the program control again.)

A thread and a task are similar and are often confused. Most computers can only execute one program instruction at a time, but because they operate so fast, they appear to run many programs and serve many users simultaneously. The computer operating system gives each program a "turn" at running, then requires it to wait while another program gets a turn. Each of these programs is viewed by the operating system as a task for which certain resources are identified and kept track of. The operating system manages each

application program in your PC system (spreadsheet, word processor, Web browser) as a separate task and lets you look at and control items on a task list. If the program initiates an I/O request, such as reading a file or writing to a printer, it creates a thread. The data kept as part of a thread allows a program to be reentered at the right place when the I/O operation completes. Meanwhile, other concurrent uses of the program are maintained on other threads. Most of today's operating systems provide support for both multitasking and multithreading. They also allow multithreading within program processes so that the system is saved the overhead of creating a new process for each thread.

The POSIX.4a C specification provides a set of application program interfaces that allow a programmer to include thread support in the program. Higher-level program development tools and application subsystems and middleware also offer thread management facilities. Languages that support object-oriented programming also accommodate and encourage multithreading in several ways. Java supports multithreading by including synchronization modifiers in the language syntax, by providing classes developed for multithreading that can be inherited by other classes, and by doing background "garbage collection" (recovering data areas that are no longer being used) for multiple threads.

>> Find white papers, products and vendors related to thread.

This word suggested by: Rene Martinez
Last updated on: Jul 26, 2003

<< Back to previous page Go to whatis.com home page >>

thread

Last modified: Thursday, October 24, 2002

(1) In online discussions, a series of messages that have been posted as replies to each other. A single forum or conference typically contains many threads covering different subjects. By reading each message in a thread, one after the other, you can see how the discussion has evolved. You can start a new thread by posting a message that is not a reply to an earlier message.

(2) In programming, a part of a program that can execute independently of other parts. Operating systems that support multithreading enable programmers to design programs whose threaded parts can execute concurrently.

•E-mail this definition to a colleague•

Related Categories

Error

internet.com
Developer
Downloads
International
Internet Lists
Internet News
Internet Resources
IT
Linux/Open Source
Personal Technology
Small Business Windows Technology
xSP Resources

Search internet.com
Advertise
Corporate Info
Newsletters
Tech Jobs
E-mail Offers

internet commerce
Be a Commerce Partner
Online College Degree
Donate Car
Price Comparison
Laptop Computers
Cheap Airline Tickets
Digital Camera Store
Internet Marketing
Promote Your Website
Dedicated Servers
Music
Online Marketing
Managed Hosting
Dedicated Server

*For internet.com pages about **thread** CLICK HERE. Also check out the following links!*

LINKS

= Great Page!

Introduction to Multithreading, Superthreading and Hyperthreading
A comprehensive explanation of these three threading techniques.

Multithreaded programming FAQ
FAQ and summary on multithreaded programming from the omp.programming.threads newsgroup.

Sun's threads page
Provides links to information on threads and multithreads, FAQs, user and man guides, example programs that use threads, and links to related sites containing papers and bibliographies.

Operating Systems

Related Terms

forum

Hyper-Threading

multithreading

mutex

online service

TechEncyclopedia More than 20,000 IT terms

Results found for: thread

thread

(1) One transaction or message in a multithreaded system. See multithreading.

(2) A topic or theme in an Internet newsgroup or groupware program that generates ongoing e-mail from interested parties. See threaded discussion.

■ TERMS SIMILAR TO YOUR ENTRY

Entries before thread

- third party
- Thomson Gale
- THOR
- Thoroughbred Basic
- thrashing

Entries after thread

- threaded connector
- threaded discussion
- threading
- threat
- threat and risk assessment

■ DEFINE ANOTHER IT TERM

Define

Or get a random definition

TechEncyclopedia More than 20,000 IT terms

Results found for: multithreading

multithreading

Multitasking within a single program. It allows multiple streams of execution to take place concurrently within the same program, each stream processing a different transaction or message. In order for a multithreaded program to achieve true performance gains, it must be run in a multitasking or multiprocessing environment, which allows multiple operations to take place.

It Depends on the Application
Certain types of applications lend themselves to multithreading. For example, in an order processing system, each order can be entered independently of the other orders. In an image editing program, a calculation-intensive filter can be performed on one image, while the user works on another. Multithreading is also used to create synchronized audio and video applications.

Symmetric Multiprocessing
A symmetric multiprocessing (SMP) operating system uses multithreading to allow multiple CPUs to be controlled at the same time. See SMP.

Reentrant Code
Multithreading generally uses reentrant code, which cannot be modified when executing, so that the same code can be shared by multiple programs. See multiprocessing and Hyper-Threading.

TERMS SIMILAR TO YOUR ENTRY

Entries before multithreading

- multisession drive
- Multistation Access Unit
- MultiSync monitor
- multitasking
- multitenant

Entries after multithreading

- multitier application
- multiuser
- Multiuser DOS
- multiuser license
- multiuser NT

DEFINE ANOTHER IT TERM

Define

Or get a random definition

MENU
Home
Term of the Day
New Terms
Pronunciation
New Links
Quick Reference
Did You Know?
Search Tool
Tech Support
Webopedia Jobs
About Us
Link to Us
Advertising

Compare Prices:
go
HardwareCentral

Talk To Us...
Submit a URL
Suggest a Term
Report an Error

multithreading

Last modified: Thursday, October 24, 2002

The ability of an operating system to execute different parts of a program, called *threads*, simultaneously. The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other.

•E-mail this definition to a colleague•

For internet.com pages about **multithreading** CLICK HERE. Also check out the following links!

LINKS = Great Page!

Introduction to Multithreading, Superthreading and Hyperthreading
A comprehensive explanation of these three threading techniques.

Related Categories
Operating Systems

Related Terms
.NET
Hyper-Threading
multitasking
SMP
thread

comp.programming.threads FAQ [last mod 97/5/24]

From: bos@serpentine.com (Bryan O'Sullivan)
Newsgroups: comp.programming.threads
Subject: comp.programming.threads FAQ [last mod 97/5/24]
Date: 20 Aug 1997 11:49:24 -0700
Message-ID: <87aficzo3v.fsf@serpentine.com>
Reply-To: threads-faq@serpentine.com
Summary: frequent topics of discussion on the threads programming news

URL: http://www.serpentine.com/~bos/threads-faq/
Posting-Frequency: monthly
Archive-name: threads-faq/part1
Last-modified: Sat May 24 21:52:47 1997

0. TABLE OF CONTENTS

2. Introduction

This posting consists of answers to many of the questions most frequently asked and summaries of the topics most frequently covere on comp.programming.threads, the Usenet newsgroup for discussion of issues in multithreaded programming. The purpose of this posting is circulate existing information, and to avoid rehashing old topics o discussion and questions. Please read all parts of this document before posting to this newsgroup.

The FAQ is posted monthly to comp.programming.threads, in multiple parts. It is also available on the World-Wide Web, at <URL: http://www.serpentine.com/~bos/threads-faq>. You may prefer t browse the FAQ on the Web rather than on Usenet, as it contains man useful hyperlinks (and tables are readable, which is unfortunately the case for the text version).

2.1. Reader contributions and comments

Your contributions, comments, and corrections are welcomed; mail se to <threads-faq@serpentine.com> will be dealt with as quickly as I manage. Generally, performing a reply or followup to this article f within your newsreader should do the Right Thing.

While I am more than happy to include submissions of material for t FAQ if they seem appropriate, it would make my life a lot easier if such text were proof-read in advance, and kept concise. I don't hav as much time as I would like to digest 15K text files and summarise them in three paragraphs for inclusion here. If you are interested contributing material, please see the to-do list at the end of part of the FAQ.

2.2. How to read this FAQ

Some headers in this FAQ are preceded by words in parentheses, such "(POSIX)". This indicates that the sections in question are specifi to a particular threads family, or to the implementation provided b specific vendor.

Wherever it may not otherwise be obvious that a particular section refers only to some families or implementations, you will find one more of the following key words to help you.

Key Implementation
DCE OSF/DCE threads (POSIX draft 4)
OS/2 IBM OS/2 threads
POSIX POSIX 1003.1c-1995 standard threads
UI Unix International threads
Unix Of general relevance to Unix users
WIN32 Microsoft Win32 API threads

2.3. Acknowledgments and caveats

Although this FAQ has been the result of a co-operative effort, any blame for inaccuracies and/or errors lies entirely with my work. I would like to thank the following people for their part in contributing to this FAQ:

Dave Butenhof <butenhof@zko.dec.com>
Bil Lewis <bil@lambdaCS.com>

3. What are threads?

A thread is an encapsulation of the flow of control in a program. M people are used to writing single-threaded programs - that is, programs that only execute one path through their code "at a time". Multithreaded programs may have several threads running through different code paths "simultaneously".

Why are some phrases above in quotes? In a typical process in which multiple threads exist, zero or more threads may actually be runnin at any one time. This depends on the number of CPUs the computer on which the process is running, and also on how the threads system is implemented. A machine with _n_ CPUs can, intuitively enough, run n more than _n_ threads in parallel, but it may give the appearance o running many more than _n_ "simultaneously", by sharing the CPUs am threads.

3.1. Why are threads interesting?

A context switch between two threads in a single process is _considerably_ cheaper than a context switch between two processes. addition, the fact that all data except for stack and registers are shared between threads makes them a natural vehicle for expressing tasks that can be broken down into subtasks that can be run cooperatively.

3.2. A little history

If you are interested in reading about the history of threads, see relevant section of the comp.os.research FAQ at <URL: http://www.serpentine.com/~bos/os-faq>.

4. What are the main families of threads?

There are two main families of threads:
* POSIX-style threads, which generally run on Unix systems.
* Microsoft-style threads, which generally run on PCs.

These families can be further subdivided.

4.1. POSIX-style threads

This family consists of three subgroups:
* "Real" POSIX threads, based on the IEEE POSIX 1003.1c-1995 (als known as the ISO/IEC 9945-1:1996) standard, part of the ANSI/IE 1003.1, 1996 edition, standard. POSIX implementations are, not surprisingly, the emerging standard on Unix systems.
  + POSIX threads are usually referred to as Pthreads.
  + You will often see POSIX threads referred to as POSIX.1c threads, since 1003.1c is the section of the POSIX standar that deals with threads.
  + You may also see references to draft 10 of POSIX.1c, which became the standard.
* DCE threads are based on draft 4 (an early draft) of the POSIX threads standard (which was originally named 1003.4a, and becam 1003.1c upon standardisation). You may find these on some Unix implementations.
* Unix International (UI) threads, also known as Solaris threads,

are based on the Unix International threads standard (a close relative of the POSIX standard). The only major Unix variants t support UI threads are Solaris 2, from Sun, and UnixWare 2, fro SCO.

Both DCE and UI threads are fairly compatible with the POSIX thread standard, although converting from either to "real" POSIX threads w require a moderate amount of work.

Those few tardy Unix vendors who do not yet ship POSIX threads implementations are expected to do so "real soon now". If you are developing multithreaded applications from scratch on Unix, you wou do well to use POSIX threads.

4.2. Microsoft-style threads

This family consists of two subgroups, both originally developed by Microsoft.

* WIN32 threads are the standard threads on Microsoft Windows 95 Windows NT.
* OS/2 threads are the standard threads on OS/2, from IBM.

Although both of these were originally implemented by Microsoft, th have diverged somewhat over the years. Moving from one to the other will require a moderate amount of work.

4.3. Others

Mach and its derivatives (such as Digital UNIX) provide a threads package called C threads. This is not very widely used.

5. Some terminology

The terms here refer to each other in a myriad of ways, so the best way to navigate through this section is to read it, and then read i again. Don't be afraid to skip forwards or backwards as the need appears.

5.1. (DCE, POSIX, UI) Async safety

Some library routines can be safely called from within signal handlers; these are referred to as async-safe. A thread that is executing some async-safe code will not deadlock if it is interrupt by a signal. If you want to make some of your own code async-safe, should block signals before you obtain any locks.

5.2. Asynchronous and blocking system calls

Most system calls, whether on Unix or other platforms, block (or "suspend") the calling thread until they complete, and continue its

execution immediately following the call. Some systems also provide asynchronous (or _non-blocking_) forms of these calls; the kernel notifies the caller through some kind of out-of-band method when su a system call has completed.

Asynchronous system calls are generally much harder for the program to deal with than blocking calls.

5.3. Context switch

A context switch is the action of switching a CPU between executing one thread and another (or transferring control between them). This may involve crossing one or more protection boundary.

5.4. Critical section

A critical section of code is one in which data that may be accesse by other threads are inconsistent. At a higher level, a critical section can be viewed as a section of code in which a guarantee you make to other threads about the state of some data may not be true.

If other threads can access these data during a critical section, y program may not behave correctly. This may cause it to crash, lock produce incorrect results, or do just about any other unpleasant th you care to imagine.

Other threads are generally denied access to inconsistent data duri a critical section (usually through use of locks). If some of your critical sections are too long, however, it may result in your code performing poorly.

5.5. Lightweight process

A lightweight process (also known in some implementations, confusingly, as a _kernel thread_) is a schedulable entity that the kernel is aware of. On most systems, it consists of some execution context and some accounting information (i.e. much less than a full-blown process).

Several operating systems allow lightweight processes to be "bound" particular CPUs; this guarantees that those threads will only execu on the specified CPUs.

5.6. MT safety

If some piece of code is described as MT-safe, this indicates that can be used safely within a multithreaded program, _and_ that it supports a "reasonable" level of concurrency. This isn't very interesting; what you, as a programmer using threads, need to worry about is code that is _not_ MT-safe. MT-unsafe code may use global

and/or static data. If you need to call MT-unsafe code from within multithreaded program, you may need to go to some effort to ensure that only one thread calls that code at any time.

Wrapping a global lock around MT-unsafe code will generally let you call it from within a multithreaded program, but since this does no permit concurrent access to that code, it is not considered to make MT-safe.

If you are trying to write MT-safe code using POSIX threads, you ne to worry about a few issues such as dealing correctly with locks across calls to fork(2) (if you are wondering what to do, read abou the pthread_atfork(3) library call).

5.7. Protection boundary

A protection boundary protects one software subsystem on a computer from another, in such a way that only data that is explicitly share across such a boundary is accessible to the entities on both sides. general, all code within a protection boundary will have access to data within that boundary.

The canonical example of a protection boundary on most modern syste is that between processes and the kernel. The kernel is protected f processes, so that they can only examine or change its internal sta in certain strictly-defined ways.

Protection boundaries also exist between individual processes on mo modern systems. This prevents one buggy or malicious process from wreaking havoc on others.

Why are protection boundaries interesting? Because transferring control across them is expensive; it takes a lot of time and work.

5.8. Scheduling

Scheduling involves deciding what thread should execute next on a particular CPU. It is usually also taken as involving the context switch to that thread.

6. What are the different kinds of threads?

There are two main kinds of threads implementations:
* User-space threads, and
* Kernel-supported threads.

There are several sets of differences between these different threa implementations.

6.1. Architectural differences

User-space threads live without any support from the kernel; they maintain all of their state in user space. Since the kernel does no know about them, they cannot be scheduled to run on multiple processors in parallel.

Kernel-supported threads fall into two classes.

* In a "pure" kernel-supported system, the kernel is responsible scheduling all threads.
* Systems in which the kernel cooperates with a user-level librar to do scheduling are known as _two-level_, or _hybrid_, systems Typically, the kernel schedules LWPs, and the user-level librar schedules threads onto LWPs.

Because of its performance problems (caused by the need to cross th user/kernel protection boundary twice for _every_ thread context switch), the former class has fewer members than does the latter (a least on Unix variants). Both classes allow threads to be run acros multiple processors in parallel.

6.2. Performance differences

In terms of context switch time, user-space threads are the fastest with two-level threads coming next (all other things being equal). However, if you have a multiprocessor, user-level threads can only run on a single CPU, while both two-level and pure kernel-supported threads can be run on multiple CPUs simultaneously.

6.3. Potential problems with functionality

Because the kernel does not know about user threads, there is a dan that ordinary blocking system calls will block the entire process (this is _bad_) rather than just the calling thread. This means tha user-space threads libraries need to jump through hoops in order to provide "blocking" system calls that don't block the entire process

This problem also exists with two-level kernel-supported threads, though it is not as acute as for user-level threads. What usually happens here is that system calls block entire LWPs. This means tha if more threads exist than do LWPs and all of the LWPs are blocked system calls, then other threads that could potentially make forwar progress are prevented from doing so.

The Solaris threads library provides a reasonable solution to this problem. If the kernel notices that all LWPs in a process are block it sends a signal to the process. This signal is caught by the user-level threads library, which can create another LWP so that th process will continue to make progress.

7. Where can I find books on threads?

There are several books available on programming with threads, with more due out in the near future. Note also that the programmer's manuals that come with most systems that provide threads packages w have sections on using those threads packages.

7.1. POSIX-style threads

David R. Butenhof, _Programming with POSIX Threads_. Addison-Wesley ISBN 0-201-63392-2.
This book gives a comprehensive and well-structured overview programming with POSIX threads, and is a good text for the working programming to work from. Detailed examples and discussions abound.

Steve Kleiman, Devang Shah and Bart Smaalders, _Programming With Threads_. SunSoft Press, ISBN 0-13-172389-8.
<URL: http://www.sun.com/smi/ssoftpress/books/Kleiman/Kleima tml>
This book goes into considerably greater depth than the othe SunSoft Press offering (see below), and is also recommended the working programmer who expects to deal with threads on a day-to-day basis. It includes many detailed examples.

Bil Lewis and Daniel J. Berg, _Threads Primer_. SunSoft Press, ISBN 0-13-443698-9.
<URL: http://www.sun.com/smi/ssoftpress/books/Lewis/Lewis.ht

This is a good introduction to programming with threads for programmers and managers. It concentrates on UI and POSIX threads, but also covers use of OS/2 and WIN32 threads.

Charles J. Northrup, _Programming With Unix Threads_. John Wiley & Sons, ISBN 0-471-13751-0.
<URL: http://www.wiley.com/compbooks/catalog/14/13751-0.html
This book details the UI threads interface, focusing mostly the Unixware implementation. This is an introductory book.

7.2. Microsoft-style threads

Jim Beveridge, Robert Wiener, _Multithreading Applications in Win32 Addison-Wesley, ISBN 0-201-44234-5.
<URL: http://www.aw.com/devpress/titles/44234.html>.
Seasoned Win32 programmers, neophytes, and programmers being dragged kicking and screaming from the Unix world are all likely to find this book a useful resource. It doubles as primer and reference on writing and debugging robust multithreaded code, and provides a thorough exposition on th subject.

Len Dorfman, Marc J. Neuberger, _Effective Multithreading with OS/2
Publisher and ISBN unknown.
This book covers the OS/2 threads API and contains many examples, but doesn't have much by way of concepts.

Thuan Q. Pham, Pankaj K. Garg, _Multithreaded Programming with Windows NT_. Prentice Hall, ISBN 0-131-20643-5.
<URL: http://www.prenhall.com/013/120642/12064-2.html>
Not surprisingly, this book focuses on WIN32 threads, but it also mentions other libraries in passing. It also deals with some relatively advanced topics, and has a thorough bibliography.

7.3. Books on implementations

If you are interested in how modern operating systems support threa and multiprocessors, there are a few excellent books available that may be of interest to you.

Curt Schimmel, _Unix Systems for Modern Architectures_.
Addison-Wesley, ISBN 0-201-63338-8.
<URL: http://www.aw.com/cp/schimmel.html>
This book gives a lucid account of the work needed to get Un (or, for that matter, more or less anything else) working on modern system that incorporates multiple processors, each wi its own cache. While it has some overlap with the Vahalia bo (see below), it has a smaller scope, and thus deals with sha topics in more detail.

Uresh Vahalia, _Unix Internals: the New Frontiers_. Prentice Hall, ISBN 0-13-101908-2.
<URL: http://www.prenhall.com/013/101907/10190-7.html>
This is the best kernel internals book currently available. deals extensively with building multithreaded kernels, implementing LWPs, and scheduling on multiprocessors. Given choice, I would buy _both_ this and the Schimmel book.

Ben Catanzaro, _Multiprocessor System Architectures_. SunSoft Press ISBN 0-13-089137-1.
<URL: http://www.sun.com/smi/ssoftpress/books/Catanzaro/Cata ro.html>
I don't know much about this book, but it deals with both th hardware and software (kernel and user) architectures used t put together modern multiprocessor systems.

7.4. The POSIX threads standard

To order ISO/IEC standard 9945-1:1996, which is also known as ANSI/IEEE POSIX 1003.1-1995 (and includes 1003.1c, the part that de with threads), you can call +1-908-981-1393. The document reference

number in the IEEE publications catalogue is SH 94352-NYF, and the price to US customers is $120 (shipping overseas costs extra).

Unless you are implementing a POSIX threads package, you should not ever need to look at the POSIX threads standard. It is the last pla you should look if you wish to learn about threads!

Neither IEEE nor ISO makes standards available for free; please do ask whether the POSIX threads standard is available on the Web. It isn't.

8. Where can I obtain training on using threads?

Organisation Contact Description
Sun Microsystems <training_seats@Sun.COM>
+1-408-276-3630 Classes at Sun and on-site classes
Lambda Computer Science
(Bil Lewis) <URL: http://www.lambdaCS.com>
+1-415-328-8952 Seminars and on-site classes
Phoenix Technologies
(Chris Crenshaw) <phnxtech@attmail.com>
+1-908-286-2118
Marc Staveley <marc@staveley.com>

9. (Unix) Are there any freely-available threads packages?

* Xavier Leroy <xleroy@inria.fr> has written a POSIX threads implementation for Linux 2.x that uses pure kernel-supported threads. While the package is currently in alpha testing, it is allegedly very stable. For more information, see <URL: http://pauillac.inria.fr/~xleroy/linuxthreads>.
* Michael T. Peterson <mtp@big.aa.net> has written a user-space POSIX and DCE threads package for Intel-based Linux systems; it called PCthreads. See <URL: http://www.aa.net/~mtp/PCthreads.ht for more information.
* Christopher Provenzano <proven@mit.edu> has written a fairly portable implementation of draft 8 of the POSIX threads standar See <URL: http://www.mit.edu:8001/people/proven/pthreads.html> further details. _Note_: as far as I can see, development of th library has halted (at least temporarily), and it still contain many serious bugs.
* Georgia Tech's OS group has a fairly portable user-level thread implementation of the Mach C threads package. It is called Cthreads, and can be found at <URL: ftp://ftp.cc.gatech.edu/pub/groups/systems/Falcon/ istribution.tar.gz>.
* Frank Müller, of the POSIX / Ada-Runtime Project (PART) has mad available an implementation of draft 6 of the POSIX 1003.4a Pthreads specification, which runs under SunOS 4, Solaris 2.x, SCO Unix, FreeBSD and Linux. For more information, see

<URL: file://ftp.cs.fsu.edu/pub/PART/PTHREADS/pthreads_ANNOUNCE
* Elan Feingold has written a threads package called ethreads; I don't know anything about it, other than that it is available f <URL: ftp://frmap711.mathp7.jussieu.fr/pub/scratch/rideau/misc/ eads/ethreads/ethreads.tgz>.
* QuickThreads is a toolkit for building threads packages, writte by David Keppel <pardo@cs.washington.edu>. It is available from <URL: ftp://ftp.cs.washington.edu/pub/qt-001.tar.Z>, with an accompanying tech report at <URL: ftp://ftp.cs.washington.edu/tr/1993/05/UW-CSE-93-05-06.PS . The code as distributed includes ports for the Alpha, x86, 88000, MIPS, SPARC, VAX, and KSR1.

10. (DCE, POSIX, UI) Why does my threaded program not handle signals

Signals and threads do not mix well. A lot of programmers start out writing their code under the mistaken assumption that they can set signal handler for each thread; this is not the way things work. Yo can _block_ or _unblock_ signals on a thread-by-thread basis, but t is not the same thing.

When it comes to dealing with signals, the best thing you can do is create a thread whose sole purpose is to handle signals for the ent process. This thread should loop calling sigwait(2); this allows it deal with signals synchronously. You should also make sure that all threads (_including_ the one that calls sigwait) have the signals y are interested in handling blocked. Handling signals synchronously this way greatly simplifies things.

Note, also, that sending signals to other threads within your own process is not a friendly thing to do, unless you are careful with signal masks. For an explanation, see the section on asynchronous cancellation.

Finally, using sigwait and installing signals handlers for the sign you are sigwaiting for is a bad idea.

11. (DCE?, POSIX) Why does everyone tell me to avoid asynchronous ca

Asynchronous cancellation of threads is, in general, evil. The reas for this is that it is usually (very) difficult to guarantee that t recipient of an asynchronous cancellation request will not be in a critical section. If a thread should die in the middle of a critica section, this will very likely cause your program to misbehave.

Code that can deal sensibly with asynchronous cancellation requests _not_ referred to as async-safe; that means something else (see the terminology section of the FAQ). You won't see much code around tha handles asynchronous cancellation requests properly, and you should try write any of your own unless you have compelling reasons to do

Deferred cancellation is your friend.

12. Why are reentrant library and system call interfaces good?

There are two approaches to providing system calls and library interfaces that will work with multithreaded programs. One is to simply wrap all the appropriate code with mutexes, thereby guaranteeing that only one thread will execute any such routine at time.

While this approach mostly works, it provides terrible performance. For functions that maintain state across multiple invocations (e.g. strtok() and friends), this approach simply doesn't work at a hence the existence of "_r" interfaces on many Unix systems (see below).

A better solution is to ensure that library calls can safely be performed by multiple threads at once.

12.1. (DCE, POSIX, UI) When should I use thread-safe "_r" library call

If your system provides threads, it will probably provide a set of thread-safe variants of standard C library routines. A small number these are mandated by the POSIX standard, and many Unix vendors provide their own useful supersets, including functions such as gethostbyname_r().

Unfortunately, the supersets that different vendors support do not necessarily overlap, so you can only _safely_ use the standard POSIX-mandated functions. The thread-safe routines are conceptually "cleaner" than their stateful counterparts, though, so it is good practice to use them wherever and whenever you can.

13. (POSIX) How can I perform a join on any thread?

UI threads allow programmers to join on any thread that happens to terminate by passing the appropriate argument to thr_join(). This i not possible under POSIX and, yes, there is a rationale behind the absence of this feature.

Unix programmers are used to being able to call wait() in such a wa that it will return when "any" process exits, but expecting this to work for threads can cause confusion for programmers trying to use threads. The important thing to note here is that Unix processes ar based around a notion of parent and child; this is a notion that is _not_ present in most threads systems. Since threads don't contain this notion, joining on "any" thread could have the undesirable eff of having the join return once a completely unrelated thread happen to exit.

In many (perhaps even most) threaded applications, you do not want be able to join with any thread in your process. Consider, for example, a library call that one of your threads might make, which its turn might start a few threads and try to join on them. If anot of your threads, joining on "any" thread, happened to join on one o the library call's threads, that would lead to incorrect program behaviour.

If you want to be able to join on any thread so that, for example, can keep track of the number of running threads, you can achieve th same functionality by starting detached threads and having them decrememnt a (suitably locked, of course) counter as they exit.

14. (DCE, UI, POSIX) After I create a certain number of threads, my crashes

By default, threads are created non-detached. You need to perform a join on each non-detached thread, or else storage will never be fre up when they exit. As an alternative, you can create detached threa for which storage will be freed as soon as they exit. This latter approach is generally better; you shouldn't create non-detached threads unless you explicitly need to know when or if they exit.

15. Where can I find POSIX thread benchmarks?

I do not know of any benchmarks.

16. Does any DBMS vendor provide a thread-safe interface?

Oracle 7.3 and above provide thread-safe database client interfaces as do Sybase System 11.1 and above, and Informix ESQL 7 and above.

If you are still using an older release of any of these products, i is possible to hack together a set of intermediate thread-safe interfaces to your database if you really need it, but this require moderately large amount of work.

17. Why is my threaded program running into performance problems?

There are many possible causes for performance problems in multithreaded programs. Given the scope for error, all I can do is detail a few common pitfalls briefly, and point you at the section this FAQ on books about multithreaded programming.

* You can probably discount the performance of the threads packag you are using almost straight away, unless it is a user-level package. If so, you may want to try to find out whether your wh process is blocking when you execute certain system calls. Otherwise, you should look at your own code unless you have a v strong reason for believing that there may be a problem with yo threads package.

* Look at the granularity of your locks. If a single lock protect too much data for which there is contention, you will needlessl serialise your code. On the other hand, if your locks protect v small amounts of data, you will spend too much time obtaining a releasing locks. If your vendor is worth their salt, they will have a set of tools available that allow you to profile your program's behaviour and look for areas of high contention for locks. Tools of this kind are invaluable.

18. What tools will help me to program with threads?

* The TNF Tools are a set of extensible tools that allow users to gather and analyse trace information from the Solaris kernel an from user processes and threads. They can be used to correlate user and kernel thread activity, and also to determine such thi as lock hold times. They are available for free from Sun; for m information, see
<URL: http://opcom.sun.ca/toolpages/tnftools.html>.
* The debugger that comes with the DevPro compiler set from Sun understands threads.
* GDB nominally understands threads, but only supports them (and a flaky way) under some versions of Irix and a few other system (mostly embedded machines).

19. What operating systems provide threads?

Vendor OS version Threads model POSIX API Notes
Digital Digital UNIX 4.0 mixed D4, 1c
Digital UNIX 3.2 kernel / bound D4
OpenVMS 7.0 (Alpha) see note 1 D4, 1c user model without patch kit
OpenVMS 7.0 (VAX) user D4, 1c
OpenVMS 6.2 user D4
HP HP/UX 10.20 _?_ _?_ not yet announced
HP/UX 10.10 user D4
IBM AIX 4.1 & 4.2 kernel D4, D7
AIX 3.5.x user DCE
OS/2 kernel DCE
Linux Linux 1.2.13 and above user / kernel 1c, DCE see free implementations for several packages
Linux 2.x kernel _n/a_ clone() interface
Microsoft Windows NT & 95 kernel DCE DCE kits layer on top of WIN32 threads
SGI Irix 6.2 mixed see note 2 sproc() is still kernel / bound
Irix 6.1 kernel / bound _n/a_ sproc() interface only
Sun Solaris 2.5 and above mixed / system / bound 1c
Solaris 2.4 mixed / system / bound D8 available from Sun only upon request
Solaris 2.x mixed / system / bound _n/a_ UI threads supported acros all releases
SunOS 4.x user _n/a_ LWP only

Threads model Meaning
user
a purely user-level threads system, with threads multiplexed atop a "traditional" Unix-style process
kernel
threads are "kernel entities" with no context switches taking place user mode

/ bound a thread may be explicitly bound to a particular processor
mixed
a mixed-mode scheduler where user threads are multiplexed across so number of LWPs

/ system threads have "system" contention scope (a user thread may permanently bound to an LWP)

/ bound an LWP may be permanently bound to a particular processor
API Meaning
_n/a_ no POSIX threads API provided
1c conforms to the POSIX 1003.1c-1995 threads API
DCE POSIX 1003.4a draft 4 API is available as part of the OSF DCE k for the platform
D4 DCE threads (or something similar) is bundled with the system
D7 POSIX 1003.4a draft 7 API (similar to the final 1003.1c standard but substantially different in some details)
D8 POSIX 1003.4a draft 8 API (identical in most respects to the 1003.1c standard, but with a few "gotchas")

1. OpenVMS 7.0 for Alpha shipped with kernel threads support disabl by default. The "mixed" threads model can be turned on by setti the MULTITHREAD sysgen parameter. With patch kit, the "mixed" o "user" model is determined by the main program binary (i.e. via the linker or the threadcp qualifier) in addition to MULTITHREA
2. SGI ships the POSIX 1003.1c API as a patch for Irix 6.2, but it will be bundled with future revisions of the OS.

20. What about other threads-related software?

* Douglas C. Schmidt <schmidt@cs.wustl.edu> has written a package called ACE (_Adaptive Communication Environment_). ACE supports multithreading on Unix and WIN32 platforms, and integrates popu IPC mechanisms (sockets, RPC, System V IPC) and a host of other features that C++ programmers will find useful. For details, se <URL: http://www.cs.wustl.edu/~schmidt/ACE.html>.

21. Where can I find other information on threads?

* The most comprehensive collection of threads-related informatio on the Web is at Sun's threads page, at <URL: http://www.sun.com/sunsoft/Products/Developer-products/si hreads>.

* IBM has a thorough treatment of AIX 4.1 threads (based on POSIX draft 7) at
  <URL: http://developer.austin.ibm.com/sdp/library/ref/about4.1/threa.html>.
* Digital has a brief overview of threads in Digital UNIX at
  <URL: http://www.unix.digital.com/unix/smp>.
* A bibliography on threads is available at
  <URL: http://liinwww.ira.uka.de/bibliography/Os/threads.html>.
* Tom Wagner <wagner@cs.umass.edu> and Don Towsley have written a introductory tutorial on programming with POSIX threads at
  <URL: http://centaurus.cs.umass.edu/~wagner/threads_html/tutori html>

21.1. Articles appearing in periodicals

* An introduction to programming with threads, at
  <URL: http://www.sun.com/sunworldonline/swol-02-1996/swol-02-th ds.html>, from SunWorld Online's February 1996 issue
* An introduction to programming with threads, at
  <URL: http://developer.austin.ibm.com/sdp/library/aixpert/nov94 xpert_nov94_intrmult.html>, from AIXpert Magazine's November 19 issue
* Porting DCE threads to AIX 4.1 (POSIX draft 7), at
  <URL: http://www.developer.ibm.com/sdp/library/aixpert/aug94/ai rt_aug94_PTHREADS.html>, from AIXpert Magazine's August 1994 is
* A less thorough introduction to programming with threads, at
  <URL: http://developer.austin.ibm.com/sdp/library/aixpert/aug95 xpert_aug95_thread.html>, from AIXpert Magazine's August 1995 issue
* Using signals with POSIX threads, at
  <URL: http://developer.austin.ibm.com/sdp/library/aixpert/aug95 xpert_aug95_signal.html>, from AIXpert Magazine's August 1995 issue

22. Notice of copyright and permissions

Related questions and answers

[Usenet FAQs | Search | Web FAQs | Documents | RFC Index]

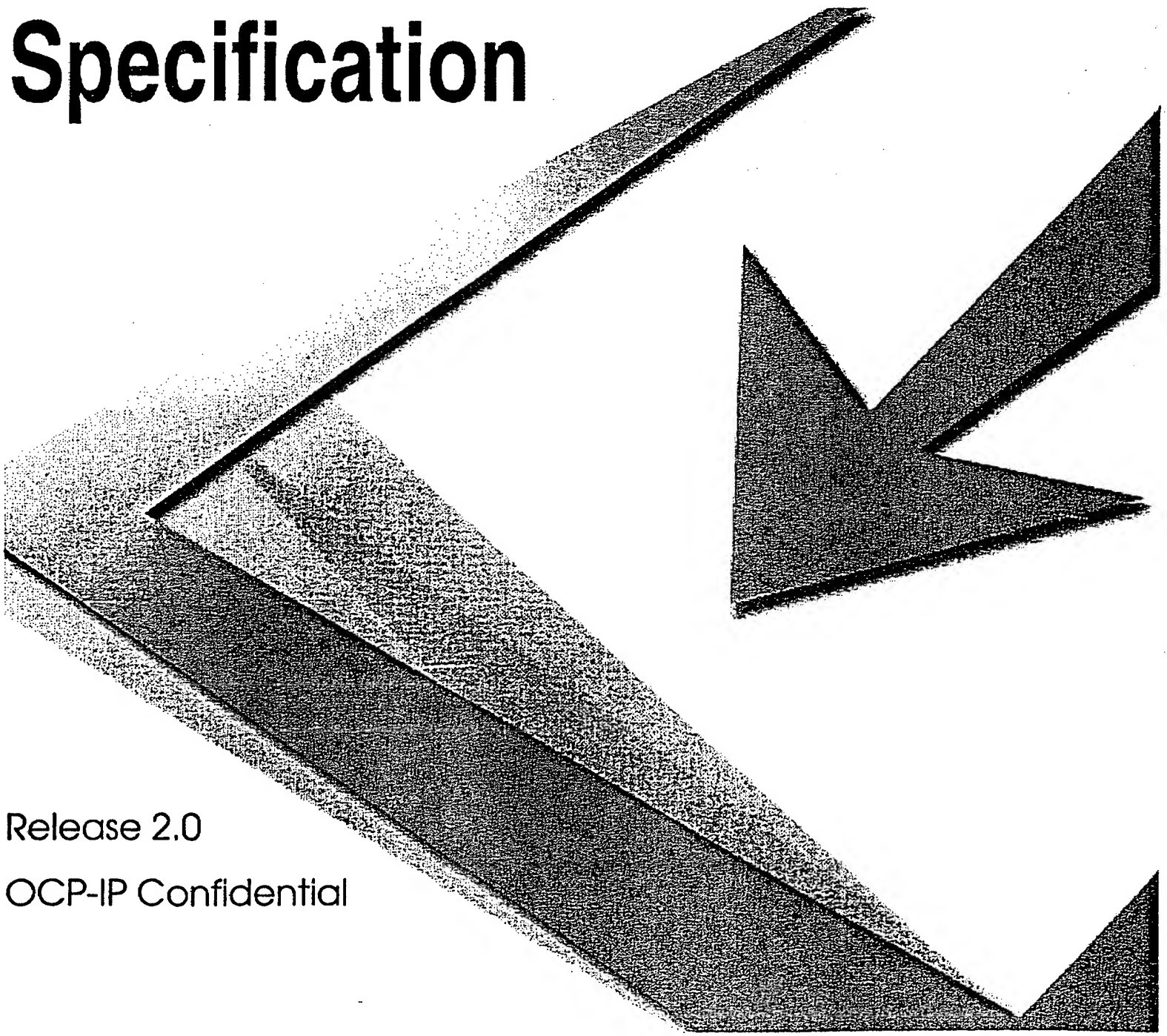
Send corrections/additions to the FAQ Maintainer:
threads-faq@serpentine.com

Last Update June 15 2004 @ 00:33 AM

OCP

International Partnership

Open Core Protocol Specification



Release 2.0

OCP-IP Confidential

Open Core Protocol Specification

Document Revision 1.1.1

Open Core Protocol Specification 2.0
Document Revision 1.1.1

Printed in the United States of America.

Part number: 161-000125-0002

EXCEPT AS OTHERWISE EXPRESSLY PROVIDED, THE OPEN CORE PROTOCOL (OCP) SPECIFICATION IS PROVIDED BY OCP-IP TO MEMBERS "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS.

OCP-IP SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND OR NATURE WHATSOEVER (INCLUDING, WITHOUT LIMITATION, ANY DAMAGES ARISING FROM LOSS OF USE OR LOST BUSINESS, REVENUE, PROFITS, DATA OR GOODWILL) ARISING IN CONNECTION WITH ANY INFRINGEMENT CLAIMS BY THIRD PARTIES OR THE SPECIFICATION, WHETHER IN AN ACTION IN CONTRACT, TORT, STRICT LIABILITY, NEGLIGENCE, OR ANY OTHER THEORY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Contents

Introduction

The Open Core Protocol™ (OCP™) delivers the only non-proprietary, openly licensed, core-centric protocol that comprehensively describes the system-level integration requirements of intellectual property (IP) cores.

While other bus and component interfaces address only the data flow aspects of core communications, the OCP unifies all inter-core communications, including sideband control and test harness signals. The OCP's synchronous unidirectional signaling produces simplified core implementation, integration, and timing analysis.

OCP eliminates the task of repeatedly defining, verifying, documenting and supporting proprietary interface protocols. The OCP readily adapts to support new core capabilities while limiting test suite modifications for core upgrades.

Clearly delineated design boundaries enable cores to be designed independently of other system cores yielding definitive, reusable IP cores with reusable verification and test suites.

Any on-chip interconnect can be interfaced to the OCP rendering it appropriate for many forms of on-chip communications:

- Dedicated peer-to-peer communications, as in many pipelined signal processing applications such as MPEG2 decoding.
- Simple slave-only applications such as slow peripheral interfaces.
- High-performance, latency-sensitive, multi-threaded applications, such as multi-bank DRAM architectures.

The OCP supports very high performance data transfer models ranging from simple request-grants through pipelined and multi-threaded objects. Higher complexity SOC communication models are supported using thread identifiers to manage out-of-order completion of multiple concurrent transfer sequences.

The CoreCreator™ tool automates the tasks of building, simulating, verifying and packaging OCP-compatible cores. IP core products can be fully "componentized" by consolidating core models, timing parameters, synthesis scripts, verification suites and test vectors in accordance with the *OCP Specification*. CoreCreator does not constrain the user to either a specific methodology or design tool.

Support

The *OCP Specification* is maintained by the Open Core Protocol International Partnership (OCP-IP™), a trade organization solely dedicated to OCP, supporting products and services. For all technical support inquiries, please contact techsupport@ocpip.org. For any other information or comments, please contact admin@ocpip.org.

Changes for Revision 2.0 of the Specification

This 2.0 revision of the *OCP Specification* is an evolution of the 1.0 version. New features were defined by the Specification Working Group of OCP-IP. The major changes and new additions are:

- Variations of the write posting model. This includes writes with responses (`writeresp_enable`) and the new command WriteNonPost.
- A new burst model that emphasizes precise bursts and bursts that have a single request with multiple data word transfers. The revision 2.0 burst model is a functional superset of the version 1.0 burst model, and replaces that burst model completely.
- To increase the flexibility of single request / multiple data bursts, support for separate write byte enables (MDataByteEnable) and a threadbusy signal for the datahandshake phase (SDataThreadBusy).
- Subsets of the OCP interface are now allowed that, for example enable read-only or write-only interfaces.
- In-band extensions that allow the master and slave to communicate core-specific in-band information (such as cacheable information or parity) with each protocol phase.
- An explicit specification of endianness. While each OCP interface by itself is endian neutral, endianness matters when communicating between cores of different data widths in a system.
- The lazy (non-blocking) synchronization pair ReadLinked/ WriteConditional was added to the existing blocking synchronization pair ReadEx/Write or WriteNonPost, to support processors that natively make use of lazy synchronization.
- A set of protocol parameters that optionally tightens the semantics of SThreadBusy and MThreadBusy, in order to guarantee that multi-threaded OCP interfaces be non-blocking.

- Reset is extended to enable dual resets, one driven by each core. This allows either core to keep the interface in the reset state until it is ready to communicate.
- Explicit defaults are specified for each parameter, reducing the number of parameters needed to fully specify a simple OCP interface. Similarly, interoperability of different OCP interfaces is simplified by specifying default tie-offs for each signal.

The 2.0 version of the *OCP Specification* is compatible with the 1.0 version with the following exceptions:

- There have been substantial changes to the burst model.
  - The restriction on byte enables for STRM bursts has been removed.
  - The definition of the `burst_aligned` parameter has been changed to remove the need for all byte enables to be turned on in every transfer.
- OCP interfaces without reset are no longer allowed.
- Ordering of datahandshake phases on multi-threaded OCP interfaces is now constrained to follow the ordering of the request phases across all threads. This constraint can be relaxed if the new SDataThreadBusy signal is used.

Document Revision 1.1.1

This version of the document has been updated with an acknowledgements page.

Acknowledgements

The following companies were instrumental in the development of the Open Core Protocol Specification, Release 2.0.

- All OCP-IP Specification Working Group members, including participants from:

  MIPS Technologies, Inc.
  Nokia Mobile Phones
  Philips Semiconductors
  Sonics, Inc.
  STMicroelectronics
  Texas Instruments Incorporated

- Other member companies providing thoughtful comments including TNI-Valiosys and LSI Logic and all additional reviewers who participated in the General Member Review

OCP-IP is pleased to recognize the historical efforts of VSIA in the development of a standard socket interface for rapid creation of interoperable virtual components.

1 Overview

The Open Core Protocol™ (OCP) defines a high-performance, bus-independent interface between IP cores that reduces design time, design risk, and manufacturing costs for SOC designs.

An IP core can be a simple peripheral core, a high-performance microprocessor, or an on-chip communication subsystem such as a wrapped on-chip bus. The Open Core Protocol:

- Achieves the goal of IP design reuse. The OCP transforms IP cores making them independent of the architecture and design of the systems in which they are used
- Optimizes die area by configuring into the OCP only those features needed by the communicating cores
- Simplifies system verification and testing by providing a firm boundary around each IP core that can be observed, controlled, and validated

The approach adopted by the Virtual Socket Interface Alliance's (VSIA) Design Working Group on On-Chip Buses (DWGOCB) is to specify a bus wrapper to provide a bus-independent Transaction Protocol-level interface to IP cores.

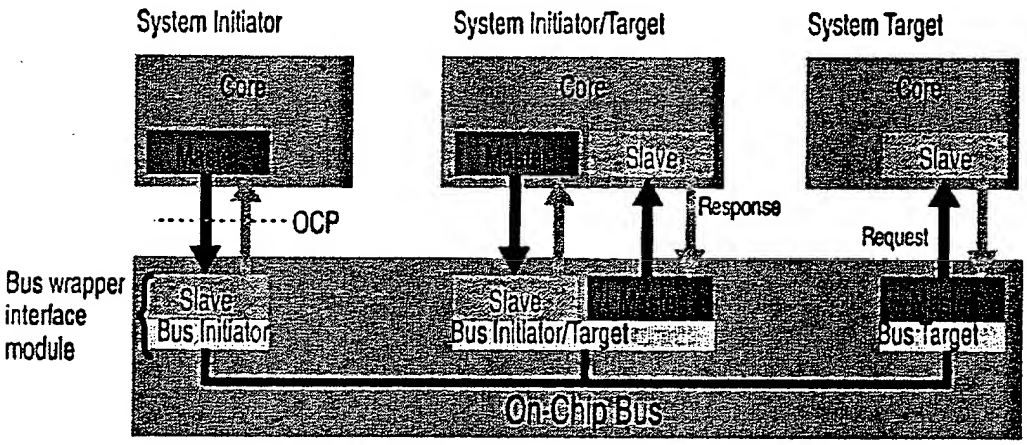
The OCP is equivalent to VSIA's Virtual Component Interface (VCI). While the VCI addresses only data flow aspects of core communications, the OCP is a superset of VCI additionally supporting configurable sideband control signaling and test harness signals. The OCP is the only standard that defines protocols to unify all of the inter-core communication.

OCP Characteristics

The OCP defines a point-to-point interface between two communicating entities such as IP cores and bus interface modules (bus wrappers). One entity acts as the master of the OCP instance, and the other as the slave. Only the master can present commands and is the controlling entity. The slave responds to commands presented to it, either by accepting data from the master, or presenting data to the master. For two entities to communicate in a peer-to-peer fashion, there need to be two instances of the OCP connecting them - one where the first entity is a master, and one where the first entity is a slave.

Figure 1 shows a simple system containing a wrapped bus and three IP core entities: one that is a system target, one that is a system initiator, and an entity that is both.

Figure 1 System Showing Wrapped Bus and OCP Instances



The characteristics of the IP core determine whether the core needs master, slave, or both sides of the OCP; the wrapper interface modules must act as the complementary side of the OCP for each connected entity. A transfer across this system occurs as follows. A system initiator (as the OCP master) presents command, control, and possibly data to its connected slave (a bus wrapper interface module). The interface module plays the request across the on-chip bus system. The OCP does not specify the embedded bus functionality. Instead, the interface designer converts the OCP request into an embedded bus transfer. The receiving bus wrapper interface module (as the OCP master) converts the embedded bus operation into a legal OCP command. The system target (OCP slave) receives the command and takes the requested action.

Each instance of the OCP is configured (by choosing signals or bit widths of a particular signal) based on the requirements of the connected entities and is independent of the others. For instance, system initiators may require more

address bits in their OCP instances than do the system targets; the extra address bits might be used by the embedded bus to select which bus target is addressed by the system initiator.

The OCP is flexible. There are several useful models for how existing IP cores communicate with one another. Some employ pipelining to improve bandwidth and latency characteristics. Others use multiple-cycle access models, where signals are held static for several clock cycles to simplify timing analysis and reduce implementation area. Support for this wide range of behavior is possible through the use of synchronous handshaking signals that allow both the master and slave to control when signals are allowed to change.

Compliance

For a core to be considered OCP compliant, it must satisfy the following conditions:

1. The core must include at least one OCP interface.
2. The core and OCP interfaces must be described using an RTL configuration file with the syntax specified in Chapter 6 on page 65.
3. Each OCP interface on the core must:
   - Comply with all aspects of the OCP interface specification
   - Have its timing described using a synthesis configuration file following the syntax specified in Chapter 7 on page 77.
4. The following practices are recommended but not required:
   a. Each non-OCP interface on the core should:

      Be described using an interface configuration file with the syntax specified in Chapter 5 on page 59.

      Have its timing described using a synthesis configuration file with the syntax specified in Chapter 7 on page 77.

   b. A performance report as specified in Chapter 12 on page 175 (or an equivalent report) should be included for the core.

Part I *Specification*

2 Theory of Operation

The Open Core Protocol interface addresses communications between the functional units (or IP cores) that comprise a system on a chip. The OCP provides independence from bus protocols without having to sacrifice high-performance access to on-chip interconnects. By designing to the interface boundary defined by the OCP, you can develop reusable IP cores without regard for the ultimate target system.

Given the wide range of IP core functionality, performance and interface requirements, a fixed definition interface protocol cannot address the full spectrum of requirements. The need to support verification and test requirements adds an even higher level of complexity to the interface. To address this spectrum of interface definitions, the OCP defines a highly configurable interface. The OCP's structured methodology includes all of the signals required to describe an IP cores' communications including data flow, control, and verification and test signals.

This chapter provides an overview of the concepts behind the Open Core Protocol, introduces the terminology used to describe the interface and offers a high-level view of the protocol.

Point-to-Point Synchronous Interface

To simplify timing analysis, physical design, and general comprehension, the OCP is composed of uni-directional signals driven with respect to, and sampled by the rising edge of the OCP clock. The OCP is fully synchronous and contains no multi-cycle timing paths. All signals other than the clock are strictly point-to-point.

Bus Independence

A core utilizing the OCP can be interfaced to any bus. A test of any bus-independent interface is to connect a master to a slave without an intervening on-chip bus. This test not only drives the specification towards a fully symmetric interface but helps to clarify other issues. For instance, device selection techniques vary greatly among on-chip buses. Some use address decoders. Others generate independent device select signals (analogous to a board level chip select). This complexity should be hidden from IP cores, especially since in the directly-connected case there is no decode/selection logic. OCP-compliant slaves receive device selection information integrated into the basic command field.

Arbitration schemes vary widely. Since there is virtually no arbitration in the directly-connected case, arbitration for any shared resource is the sole responsibility of the logic on the bus side of the OCP. This permits OCP-compliant masters to pass a command field across the OCP that the bus interface logic converts into an arbitration request sequence.

Commands

There are two basic commands, Read and Write and five command extensions. The WriteNonPost and Broadcast commands have semantics that are similar to the Write command. A WriteNonPost explicitly instructs the slave not to post a write. For the Broadcast command, the master indicates that it is attempting to write to several or all remote target devices that are connected on the other side of the slave. As such, Broadcast is typically useful only for slaves that are in turn a master on another communication medium (such as an attached bus).

The other command extensions, ReadExclusive, ReadLinked and WriteConditional, are used for synchronization between system initiators. ReadExclusive is paired with Write or WriteNonPost, and has blocking semantics. ReadLinked, used in conjunction with WriteConditional has non-blocking (lazy) semantics. These synchronization primitives correspond to those available natively in the instruction sets of different processors.

Address/Data

Wide widths, characteristic of shared on-chip address and data buses, make tuning the OCP address and data widths essential for area-efficient implementation. Only those address bits that are significant to the IP core should cross the OCP to the slave. The OCP address space is flat and composed of 8-bit bytes (octets).

To increase transfer efficiencies, many IP cores have data field widths significantly greater than an octet. The OCP supports a configurable data width to allow multiple bytes to be transferred simultaneously. The OCP refers to the chosen data field width as the *word size* of the OCP. The term word is used in the traditional computer system context; that is, a *word* is the natural transfer unit of the block. OCP supports word sizes of power-of-two and non-power-of-two as would be needed for a 12-bit DSP core. The OCP address is a byte address that is word aligned.

Transfers of less than a full word of data are supported by providing byte enable information that specifies which octets are to be transferred. Byte enables are linked to specific data bits (byte lanes). Byte lanes are not associated with particular byte addresses. This makes the OCP endian-neutral, able to support both big and little-endian cores.

Pipelining

The OCP allows pipelining of transfers. To support this feature, the return of read data and the provision of write data may be delayed after the presentation of the associated request.

Response

The OCP separates requests from responses. A slave can accept a command request from a master on one cycle and respond in a later cycle. The division of request from response permits pipelining. The OCP provides the option of having responses for Write commands, or completing them immediately without an explicit response.

Burst

To provide high transfer efficiency, burst support is essential for many IP cores. The extended OCP supports annotation of transfers with burst information. Bursts can either include addressing information for each successive command (which simplifies the requirements for address sequencing/burst count processing in the slave), or include addressing information only once for the entire burst.

In-band Information

Cores can pass core-specific information in-band in company with the other information being exchanged. In-band extensions exist for requests and responses, as well as read and write data. A typical use of in-band extensions is to pass cacheable information or data parity.

Threads and Connections

To support concurrency and out-of-order processing of transfers, the extended OCP supports the notion of multiple threads. Transactions within different threads have no ordering requirements, and so can be processed out of order. Within a single thread of data flow, all OCP transfers must remain ordered.

While the notion of a thread is a local concept between a master and a slave communicating over an OCP, it is possible to globally pass thread information from initiator to target using connection identifiers. Connection information helps to identify the initiator and determine priorities or access permissions at the target.

Interrupts, Errors, and other Sideband Signaling

While moving data between devices is a central requirement of on-chip communication systems, other types of communications are also important. Different types of control signaling are required to coordinate data transfers (for instance, high-level flow control) or signal system events (such as interrupts). Dedicated point-to-point data communication is sometimes required. Many devices also require the ability to notify the system of errors that may be unrelated to address/data transfers.

The OCP refers to all such communication as *sideband* (or out-of-band) signaling, since it is not directly related to the protocol state machines of the dataflow portion of the OCP. The OCP provides support for such signals through sideband signaling extensions.

Errors are reported across the OCP using two mechanisms. The error response code in the response field describes errors resulting from OCP transfers that provide responses. Write-type commands without responses cannot use the in-band reporting mechanism. The second method for reporting errors across the OCP uses out-of band error fields. These signals report more generic sideband errors, including those associated with posted write commands.

3 Signals and Encoding

OCP interface signals are grouped into dataflow, sideband, and test signals. The dataflow signals are divided into basic signals, simple extensions, burst extensions, and thread extensions. A small set of the signals from the basic dataflow group is required in all OCP configurations. Optional signals can be configured to support additional core communication requirements. All sideband and test signals are optional.

The OCP is a synchronous interface with a single clock signal. All OCP signals are driven with respect to, and sampled by, the rising edge of the OCP clock. Except for clock, OCP signals are strictly point-to-point and uni-directional. The complete set of OCP signals is shown in Figure 4 on page 29.

Dataflow Signals

The dataflow signals consist of a small set of required signals and a number of optional signals that can be configured to support additional core communication requirements. The dataflow signals are grouped into basic signals, simple extensions (which add such options as byte enables and in-band information), burst extensions (which add support for bursting), and thread extensions (which add multi-threading support).

The naming conventions for dataflow signals use the prefix M for signals driven by the OCP master and S for signals driven by the OCP slave.

Basic Signals

Table 1 lists the basic OCP signals. Only Clk and MCmd are required. The remaining OCP signals are optional.

Table 1 Basic OCP Signals

Name	Width	Driver	Function
Clk	1	*varies*	OCP clock
MAddr	configurable	master	Transfer address
MCmd	3	master	Transfer command
MData	configurable	master	Write data
MDataValid	1	master	Write data valid
MRespAccept	1	master	Master accepts response
SCmdAccept	1	slave	Slave accepts transfer
SData	configurable	slave	Read data
SDataAccept	1	slave	Slave accepts write data
SResp	2	slave	Transfer response

Clk

Clock signal for the OCP. All interface signals are synchronous to the rising edge of Clk. Clk is driven by a third entity and serves as an input to both the master and the slave.

MAddr

The Transfer address, MAddr specifies the slave-dependent address of the resource targeted by the current transfer. To configure this field into the OCP, use the `addr` parameter. To configure the width of this field, use the `addr_wdth` parameter.

MAddr is a byte address that must be aligned to the OCP word size (`data_wdth`). If the OCP word size is larger than a single byte, the aggregate is addressed at the OCP word-aligned address and the lowest

order address bits are hardwired to 0. If the OCP word size is not a power-of-2, the address is the same as it would be for an OCP interface with a word size equal to the next larger power-of-2.

MCmd
Transfer command. This signal indicates the type of OCP transfer the master is requesting. Each non-idle command is either a read or write type request, depending on the direction of data flow. Commands are encoded as follows.

Table 2 Command Encoding

MCmd[2:0]			Command	Mnemonic	Request Type
0	0	0	Idle	IDLE	(none)
0	0	1	Write	WR	write
0	1	0	Read	RD	read
0	1	1	ReadEx	RDEX	read
1	0	0	ReadLinked	RDL	read
1	0	1	WriteNonPost	WRNP	write
1	1	0	WriteConditional	WRC	write
1	1	1	Broadcast	BCST	write

The set of allowable commands can be limited using the `write_enable`, `read_enable`, `readex_enable`, `writenonpost_enable`, `rdlwrc_enable`, and `broadcast_enable` parameters as described in "Protocol Options" on page 47.

MData
Write data. This field carries the write data from the master to the slave. The field is configured into the OCP using the `mdata` parameter and its width is configured using the `data_wdth` parameter. The width is not restricted to multiples of 8.

MDataValid
Write data valid. When set to 1, this bit indicates that the data on the MData field is valid. Use the `datahandshake` parameter to configure this field into the OCP.

MRespAccept
Master response accept. The master indicates that it accepts the current response from the slave with a value of 1 on the MRespAccept signal. Use the `respaccept` parameter to enable this field into the OCP.

SCmdAccept
Slave accepts transfer. A value of 1 on the SCmdAccept signal indicates that the slave accepts the master's transfer request. To configure this field into the OCP, use the `cmdaccept` parameter.

SData
: This field carries the requested read data from the slave to the master. The field is configured into the OCP using the `sdata` parameter and its width is configured using the `data_wdth` parameter. The width is not restricted to multiples of 8.

SDataAccept
: Slave accepts write data. The slave indicates that it accepts pipelined write data from the master with a value of 1 on SDataAccept. This signal is meaningful only when datahandshake is in use. Use the `dataaccept` parameter to configure this field into the OCP.

SResp
: Response field from the slave to a transfer request from the master. The field is configured into the OCP using the `resp` parameter. Response encoding is as follows.

Table 3 Response Encoding

SResp[1:0]		Response	Mnemonic
0	0	No response	NULL
0	1	Data valid / accept	DVA
1	0	Request failed	FAIL
1	1	Response error	ERR

Simple Extensions

Table 4 lists the simple OCP extensions. The extensions add to the OCP interface address spaces, byte enables, and additional core-specific information for each phase.

Table 4 Simple OCP Extensions

Name	Width	Driver	Function
MAddrSpace	configurable	master	Address space
MByteEn	configurable	master	Request phase byte enables
MDataByteEn	configurable	master	Datahandshake phase write byte enables
MDataInfo	configurable	master	Additional information transferred with the write data
MReqInfo	configurable	master	Additional information transferred with the request
SDataInfo	configurable	slave	Additional information transferred with the read data
SRespInfo	configurable	slave	Additional information transferred with the response

MAddrSpace

This field specifies the address space and is an extension of the MAddr field that is used to indicate the address region of a transfer. Examples of address regions are the register space versus the regular memory space of a slave or the user versus supervisor space for a master.

The MAddrSpace field is configured into the OCP using the addrspace parameter. The width of the MAddrSpace field is configured with the addrspace_wdth parameter. While the encoding of the MAddrSpace field is core-specific, it is recommended that slaves use 0 to indicate the internal register space.

MByteEn

Byte enables. This field indicates which bytes within the OCP word are part of the current transfer. See "Partial Word Transfers" on page 40 for more detail on request and datahandshake phase byte enables and their relationship. There is one bit in MByteEn for each byte in the OCP word. Setting MByteEn[*n*] to 1 indicates that the byte associated with data wires [(8*n* + 7):8*n*] should be transferred. The MByteEn field is configured into the OCP using the byteen parameter and is allowed only if data_wdth is a multiple of 8 (that is, the data width is an integer number of bytes).

The allowable patterns on MByteEn can be limited using the force_aligned parameter as described on page 48.

MDataByteEn

Write byte enables. This field indicates which bytes within the OCP word are part of the current write transfer. See "Partial Word Transfers" on page 40 for more detail on request and datahandshake phase byte enables and their relationship. There is one bit in MDataByteEn for each byte in the OCP word. Setting MDataByteEn[*n*] to 1 indicates that the byte associated with MData wires [(8*n* + 7):8*n*] should be transferred. The MDataByteEn field is configured into the OCP using the mdatabyteen parameter. Setting mdatabyteen to 1 is only allowed if datahandshake_enable is 1, and only if data_wdth is a multiple of 8 (that is, the data width is an integer number of bytes).

The allowable patterns on MDataByteEn can be limited using the force_aligned parameter as described on page 48.

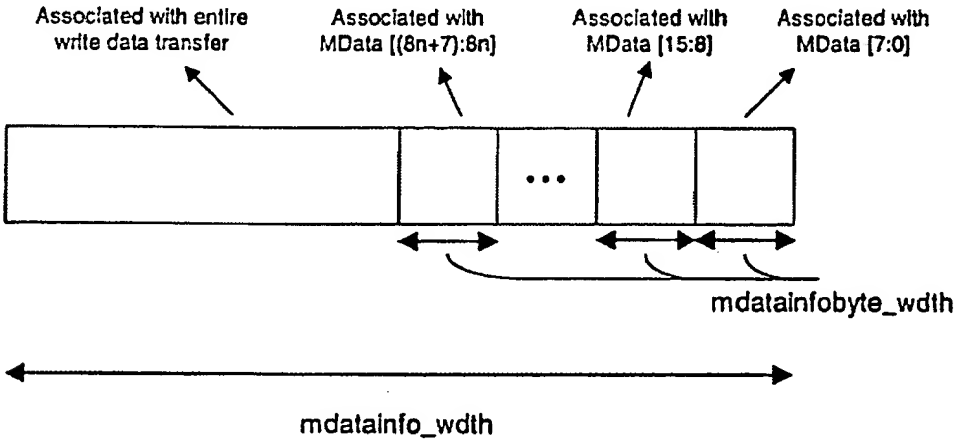
MDataInfo

Extra information sent with the write data. The master uses this field to send additional information sequenced with the write data. The encoding of the information is core-specific. To be interoperable with masters that do not provide this signal, design slaves to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 12 on page 25. Sample uses are data byte parity or error correction code values. Use the mdatainfo parameter to configure this field into the OCP, and the mdatainfo_wdth parameter to configure its width.

This field is divided in two: the low-order bits are associated with each data byte, while the high-order bits are associated with the entire write data transfer. The number of bits to associate with each data byte is

configured using the mdatainfobyte_wdth parameter. The low-order mdatainfobyte_wdth bits of MDataInfo are associated with the MData[7:0] byte, and so on.

Figure 2 MDataInfo Field



MReqInfo

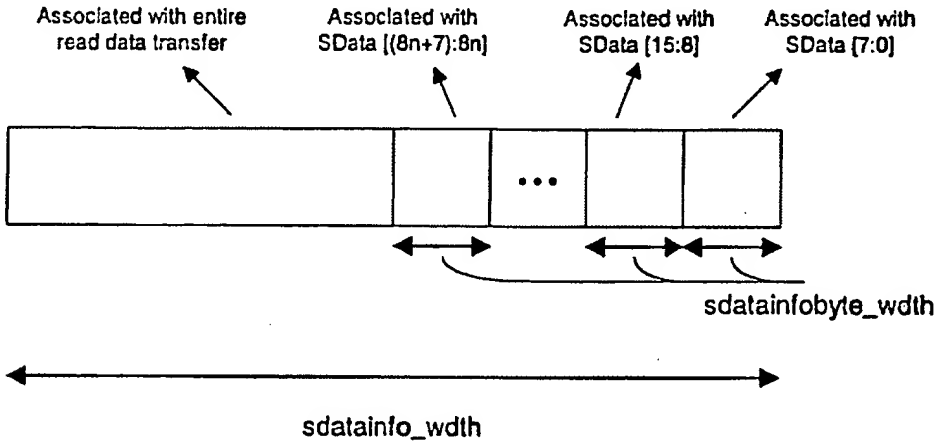
Extra information sent with the request. The master uses this field to send additional information sequenced with the request. The encoding of the information is core-specific. To be interoperable with masters that do not provide this signal, design slaves to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 12 on page 25. Sample uses are cacheable storage attributes or other access mode information. Use the reqinfo parameter to configure this field into the OCP, and the reqinfo_wdth parameter to configure its width.

SDataInfo

Extra information sent with the read data. The slave uses this field to send additional information sequenced with the read data. The encoding of the information is core-specific. To be interoperable with slaves that do not provide this signal, design masters to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 12 on page 25. Sample uses are data byte parity or error correction code values. Use the sdatainfo parameter to configure this field into the OCP, and the sdatainfo_wdth parameter to configure its width.

This field is divided into two pieces: the low-order bits are associated with each data byte, while the high-order bits are associated with the entire read data transfer. The number of bits to associate with each data byte is configured using the sdatainfobyte_wdth parameter. The low-order sdatainfobyte_wdth bits of SDataInfo are associated with the SData[7:0] byte, and so on.

Figure 3 SDataInfo Field



SRespInfo

Extra information sent with the response. The slave uses this field to send additional information sequenced with the response. The encoding of the information is core-specific. To be interoperable with slaves that do not provide this signal, design masters to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 12 on page 25. Sample uses are status or error information such as FIFO full or empty indications. Use the `respinfo` parameter to configure this field into the OCP, and the `respinfo_wdth` parameter to configure its width.

Burst Extensions

Table 5 lists the OCP burst extensions. The burst extensions allow the grouping of multiple transfers that have a defined address relationship.

Table 5 OCP Burst Extensions

Name	Width	Driver	Function
MAtomicLength	configurable	master	Length of atomic burst
MBurstLength	configurable	master	Burst length
MBurstPrecise	1	master	Given burst length is precise
MBurstSeq	3	master	Address sequence of burst
MBurstSingleReq	1	master	Burst uses single request/ multiple data protocol
MDataLast	1	master	Last write data in burst
MReqLast	1	master	Last request in burst
SRespLast	1	slave	Last response in burst

MAtomicLength
This field indicates the minimum number of transfers within a burst that are to be kept together as an atomic unit when interleaving requests from different initiators onto a single thread at the target. To configure this field into the OCP, use the `atomiclength` parameter. To configure the width of this field, use the `atomiclength_wdth` parameter. A binary encoding of the number of transfers is used. A value of 0 is not a legal encoding for MAtomicLength.

MBurstLength
The number of transfers in a burst. For precise bursts, the value indicates the total number of transfers in the burst, and is constant throughout the burst. For imprecise bursts, the value indicates the best guess of the number of transfers remaining (including the current request), and may change with every request. To configure this field into the OCP, use the `burstlength` parameter. To configure the width of this field, use the `burstlength_wdth` parameter. A binary encoding of the number of transfers is used. A value of 0 is not a legal encoding for MBurstLength.

MBurstPrecise
This field indicates whether the precise length of a burst is known at the start of the burst or not. When set to 1, `MBurstLength` indicates the precise length of the burst during the first request of the burst. To configure this field into the OCP, use the `burstprecise` parameter. When set to 0, MBurstLength for each request is a hint of the remaining burst length.

MBurstSeq
This field indicates the sequence of addresses for requests in a burst. To configure this field into the OCP, use the `burstseq` parameter. The encodings of the MBurstSeq field are shown in Table 6. The precise definition of the address sequences is described in "Burst Address Sequence" on page 42.

Table 6 MBurstSeq Encoding

MBurstSeq[2:0]			Burst Sequence	Mnemonic
0	0	0	Incrementing	INCR
0	0	1	Custom (packed)	DFLT1
0	1	0	Wrapping	WRAP
0	1	1	Custom (not packed)	DFLT2
1	0	0	Exclusive OR	XOR
1	0	1	Streaming	STRM
1	1	0	Unknown	UNKN
1	1	1	Reserved	

MBurstSingleReq
The burst has a single request with multiple data transfers. This field indicates whether the burst has a request per data transfer, or a single

request for all data transfers. To configure this field into the OCP, use the `burstsinglereq` parameter. When this field is set to 0, there is a one-to-one association of requests to data transfers; when set to 1, there is a single request for all data transfers in the burst.

MDataLast
: Last write data in a burst. This field indicates whether the current write data transfer is the last in a burst. To configure this field into the OCP, use the `datalast` parameter with `datahandshake` set to 1. When this field is set to 0, more write data transfers are coming for the burst; when set to 1, the current write data transfer is the last in the burst.

MReqLast
: Last request in a burst. This field indicates whether the current request is the last in this burst. To configure this field into the OCP, use the `reqlast` parameter. When this field is set to 0, more requests are coming for this burst; when set to 1, the current request is the last in the burst.

SRespLast
: Last response in a burst. This field indicates whether the current response is the last in this burst. To configure this field into the OCP, use the `resplast` parameter. When the field is set to 0, more responses are coming for this burst; when set to 1, the current response is the last in the burst.

Thread Extensions

Table 7 shows a list of OCP thread extensions. The extensions add support for multi-threading of the OCP interface.

Table 7 OCP Thread Extensions

Name	Width	Driver	Function
MConnID	configurable	master	Connection identifier
MDataThreadID	configurable	master	Write data thread identifier
MThreadBusy	configurable	master	Master thread busy
MThreadID	configurable	master	Request thread identifier
SDataThreadBusy	configurable	slave	Slave write data thread busy
SThreadBusy	configurable	slave	Slave request thread busy
SThreadID	configurable	slave	Response thread identifier

MConnID
: Connection identifier. This variable-width field provides the binary encoded connection identifier associated with the current transfer request.

  To configure this field use the `connid` parameter. The field width is configured with the `connid_wdth` parameter.

MDataThreadID
: Write data thread identifier. This variable-width field provides the thread identifier associated with the current write data. The field carries the binary-encoded value of the thread identifier.

  MDataThreadID is required if threads is greater than 1 and the datahandshake parameter is set to 1. MDataThreadID has the same width as MThreadID and SThreadID.

MThreadBusy
: Master thread busy. The master notifies the slave that it cannot accept any responses associated with certain threads. The MThreadBusy field is a vector (one bit per thread). A value of 1 on any given bit indicates that the thread associated with that bit is busy. Bit 0 corresponds to thread 0, and so on. The width of the field is set using the threads parameter. It is legal to enable a one-bit MThreadBusy interface for a single-threaded OCP. To configure this field, use the mthreadbusy parameter.

MThreadID
: Request thread identifier. This variable-width field provides the thread identifier associated with the current transfer request. If threads is greater than 1, this field is enabled. The field width is the next whole integer of $\log_2$(threads).

SDataThreadBusy
: Slave write data thread busy. The slave notifies the master that it cannot accept any new datahandshake phases associated with certain threads. The SDataThreadBusy field is a vector, one bit per thread. A value of 1 on any given bit indicates that the thread associated with that bit is busy. Bit 0 corresponds to thread 0, and so on.

  The width of the field is set using the threads parameter. It is legal to enable a one-bit SDataThreadBusy interface for a single-threaded OCP. To configure this field, use the sdatathreadbusy parameter.

SThreadID
: Response thread identifier. This variable-width field provides the thread identifier associated with the current transfer response. If threads is greater than 1, this field is enabled. The field width is the next whole integer of $\log_2$(threads).

SThreadBusy
: Slave thread busy. The slave notifies the master that it cannot accept any new requests associated with certain threads. The SThreadBusy field is a vector, one bit per thread. A value of 1 on any given bit indicates that the thread associated with that bit is busy. Bit 0 corresponds to thread 0, and so on. The width of the field is set using the threads parameter. It is legal to enable a one-bit SThreadBusy interface for a single-threaded OCP. To configure this field, use the sthreadbusy parameter.

Sideband Signals

Sideband signals are OCP signals that are not part of the dataflow phases, and so can change asynchronously with the request/response flow but are still synchronous to the rising edge of Clk. Sideband signals convey control information such as reset, interrupt, error, and core-specific flags. They also exchange control and status information between a core and an attached system. All sideband signals are optional.

Table 8 shows a list of the sideband extensions to the OCP.

Table 8 Sideband OCP Signals

Name	Width	Driver	Function
MError	1	master	Master Error
MFlag	configurable	master	Master flags
MReset_n	1	master	Synchronous master reset
SError	1	slave	Slave error
SFlag	configurable	slave	Slave flags
SInterrupt	1	slave	Slave interrupt
SReset_n	1	slave	Synchronous slave reset
Control	configurable	system	Core control information
ControlBusy	1	core	Hold control information
ControlWr	1	system	Control information has been written
Status	configurable	core	Core status information
StatusBusy	1	core	Status information is not consistent
StatusRd	1	system	Status information has been read

Reset, Interrupt, Error, and Core-Specific Flag Signals

MError

Master error. When the MError signal is set to 1, the master notifies the slave of an error condition. The MError field is configured with the `merror` parameter.

MFlag

Master flags. This variable-width set of signals allows the master to communicate out-of-band information to the slave. Encoding is completely core-specific.

To configure this field into the OCP, use the `mflag` parameter. To configure the width of this field, use the `mflag_wdth` parameter.

MReset_n

Synchronous master reset. The MReset_n signal is active low, as shown in Table 9. The MReset_n field is enabled by the `mreset` parameter.

Table 9 MReset Signal

MReset_n	Function
0	Reset Active
1	Reset Inactive

SError
: Slave error. With a value of 1 on the SError signal the slave indicates an error condition to the master. The SError field is configured with the `serror` parameter.

SFlag
: Slave flags. This variable-width set of signals allows the slave to communicate out-of-band information to the master. Encoding is completely core-specific.

  To configure this field into the OCP, use the `sflag` parameter. To configure the width of this field, use the `sflag_wdth` parameter.

SInterrupt
: Slave interrupt. The slave may generate an interrupt with a value of 1 on the `SInterrupt` signal. The SInterrupt field is configured with the `interrupt` parameter.

SReset_n
: Synchronous slave reset. The SReset_n signal is active low, as shown in Table 10. The SReset_n field is enabled by the `sreset` parameter.

Table 10 SReset Signal

SReset_n	Function
0	Reset Active
1	Reset Inactive

Control and Status Signals

The remaining sideband signals are designed for the exchange of control and status information between an IP core and the rest of the system. They make sense only in this environment, regardless of whether the IP core acts as a master or slave across the OCP interface.

Control
: Core control information. This variable-width field allows the system to drive control information into the IP core. Encoding is core-specific.

  Use the `control` parameter to configure this field into the OCP. Use the `control_wdth` parameter to configure the width of this field.

ControlBusy
: Core control busy. When this signal is set to 1, the core tells the system to hold the control field value constant. Use the `controlbusy` parameter to configure this field into the OCP.

ControlWr
: Core control event. This signal is set to 1 by the system to indicate that control information is written by the system. Use the `controlwr` parameter to configure this field into the OCP.

Status
: Core status information. This variable-width field allows the IP core to report status information to the system. Encoding is core-specific.

  Use the `status` parameter to configure this field into the OCP. Use the `status_wdth` parameter to configure the width of this field.

StatusBusy
: Core status busy. When this signal is set to 1, the core tells the system to disregard the status field because it may be inconsistent. Use the `statusbusy` parameter to configure this field into the OCP.

StatusRd
: Core status event. This signal is set to 1 by the system to indicate that status information is read by the system. To configure this field into the OCP, use the `statusrd` parameter.

Test Signals

The test signals add support for scan, clock control, and IEEE 1149.1 (JTAG). All test signals are optional.

Table 11 Test OCP Signals

Name	Width	Driver	Function
Scanctrl	configurable	system	Scan control signals
Scanin	configurable	system	Scan data in
Scanout	configurable	core	Scan data out
ClkByp	1	system	Enable clock bypass mode
TestClk	1	system	Test clock
TCK	1	system	Test clock
TDI	1	system	Test data in
TDO	1	core	Test data out
TMS	1	system	Test mode select
TRST_N	1	system	Test reset

Scan Interface

The Scanctrl, Scanin, and Scanout signals together form a scan interface into a given IP core.

Scanctrl
: Scan mode control signals. Use this variable width field to control the scan mode of the core. Set `scanport` to 1 to configure this field into the OCP interface. Use the `scanctrl_wdth` parameter to configure the width of this field.

Scanin
: Scan data in for a core's scan chains. Use the `scanport` parameter, to configure this field into the OCP interface and `scanport_wdth` to control its width.

Scanout
: Scan data out from the core's scan chains. Use the `scanport` parameter to configure this field into the OCP interface and `scanport_wdth` to control its width.

Clock Control Interface

The ClkByp and TestClk signals together form the clock control interface into a given IP core. This interface is used to control the core's clocks during scan operation.

ClkByp
: Enable clock bypass signal. When set to 1, this signal instructs the core to bypass the external clock source and use TestClk instead. Use the `clkctrl_enable` parameter to configure this signal into the OCP interface.

TestClk
: A gated test clock. This clock becomes the source clock when ClkByp is asserted during scan operations. Use the `clkctrl_enable` parameter to configure this signal into the OCP interface.

Debug and Test Interface

The TCK, TDI, TDO, TMS, and TRST_N signals together form an IEEE 1149 debug and test interface for the OCP.

TCK
: Test clock as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TDI
: Test data in as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TDO
: Test data out as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TMS
: Test mode select as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TRST_N
: Test logic reset signal. This is an asynchronous active low reset as defined by IEEE 1149.1. Use the `jtagtrst_enable` parameter to add this signal to the OCP interface.

Signal Configuration

The set of signals making up the OCP interface is configurable to match the characteristics of the IP core. Throughout this chapter, configuration parameters were mentioned that control the existence and width of various OCP fields. Table 12 on page 25 summarizes the configuration parameters, sorted by interface signal group. For each signal, the table also specifies the default constant tie-off, which is the inferred value of the signal if it is not configured into the OCP interface and if no other constant tie-off is specified.

For the MRespAccept, SCmdAccept, SDataAccept, MReset_n, SReset_n, ControlBusy, and StatusBusy signals, the default tie-off is also the only legal tie-off.

Table 12 OCP Signal Configuration Parameters

Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Basic	Clk	Required	Fixed	n/a
	MAddr	addr	addr_wdth	0
	MCmd	Required	Fixed	n/a
	MData	mdata	data_wdth	0
	MDataValid	datahandshake	Fixed	n/a
	MRespAccept[1]	respaccept	Fixed	1
	SCmdAccept	cmdaccept	Fixed	1
	SData[1]	sdata	data_wdth	0
	SDataAccept[2]	dataaccept	Fixed	1
	SResp	resp	Fixed	Null
Simple	MAddrSpace	addrspace	addrspace_wdth	0
	MByteEn[3]	byteen	data_wdth	all 1s
	MDataByteEn[4]	mdatabyteen	data_wdth	all 1s
	MDataInfo	mdatainfo	mdatainfo_wdth[5]	0
	MReqInfo	reqinfo	reqinfo_wdth	0

Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
	SDataInfo[1]	sdatainfo	sdatainfo_wdth[6]	0
	SRespInfo[1]	respinfo	respinfo_wdth	0
Burst	MAtomicLength	atomiclength	atomiclength_wdth	1
	MBurstLength	burstlength	burstlength_wdth	1
	MBurstPrecise	burstprecise	Fixed	1
	MBurstSeq	burstseq	Fixed	INCR
	MBurstSingleReq[7]	burstsinglereq	Fixed	0
	MDataLast[8]	datalast	Fixed	n/a
	MReqLast	reqlast	Fixed	n/a
	SRespLast[1]	resplast	Fixed	n/a
Thread	MConnID	connid	connid_wdth	0
	MDataThreadID[9]	threads>1 and datahandshake	threads	0
	MThreadBusy[10]	mthreadbusy	threads	0
	MThreadID	threads>1	threads	0
	SDataThreadBusy[11]	sdatathreadbusy	threads	0
	SThreadBusy[12]	sthreadbusy	threads	0
	SThreadID	threads>1 and resp	threads	0
Sideband	Control	control	control_wdth	0
	ControlBusy[13]	controlbusy	Fixed	0
	ControlWr[14]	controlwr	Fixed	n/a
	MError	merror	Fixed	0
	MFlag	mflag	mflag_wdth	0
	MReset_n	mreset	Fixed	1
	SError	serror	Fixed	0
	SFlag	sflag	sflag_wdth	0
	SInterrupt	Interrupt	Fixed	0
	SReset_n	sreset	Fixed	1
	Status	status	status_wdth	0
	StatusBusy[15]	statusbusy	Fixed	0
	StatusRd[16]	statusrd	Fixed	n/a

Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Test	ClkByp	clkctrl_enable	Fixed	n/a
	Scanctrl	scanport	scanctrl_wdth	n/a
	Scanin	scanport	scanport_wdth	n/a
	Scanout	scanport	scanport_wdth	n/a
	TCK	jtag_enable	Fixed	n/a
	TDI	jtag_enable	Fixed	n/a
	TDO	jtag_enable	Fixed	n/a
	TestClk	clkctrl_enable	Fixed	n/a
	TMS	jtag_enable	Fixed	n/a
	TRST_N	jtagtrst_enable	Fixed	n/a

1 MRespAccept, SData, SDataInfo, SRespInfo, and SRespLast may be included only if the `resp` parameter is set to 1.

2 SDataAccept can be included only if `datahandshake` is set to 1.

3 MByteEn has a width of `data_wdth/8` and can only be included when either `mdata` or `sdata` is set to 1 and `data_wdth` is an integer multiple of 8.

4 MDataByteEn has a width of `data_wdth/8` and can only be included when `mdata` is set to 1, `datahandshake` is set to 1, and `data_wdth` is an integer multiple of 8.

5 `mdatainfo_wdth` must be greater than or equal to `mdatainfobyte_wdth * data_wdth/8` and can be used only if `data_wdth` is a multiple of 8. `mdatainfobyte_wdth` specifies the partitioning of `MDataInfo` into transfer-specific and per-byte fields.

6 `sdatainfo_wdth` must be greater than or equal to `sdatainfobyte_wdth * data_wdth/8` and can be used only if `data_wdth` is a multiple of 8. `sdatainfobyte_wdth` specifies the partitioning of SDataInfo into transfer-specific and per-byte fields.

7 If any write-type commands are enabled, MBurstSingleReq can only be included when `datahandshake` is set to 1.

8 MDataLast can be included only if `datahandshake` is set to 1.

9 MDataThreadID is included if `threads` is greater than 1 and the `datahandshake` parameter is set to 1.

10 MThreadBusy has a width equal to `threads`. It may be included for single-threaded OCP interfaces.

11 SDataThreadBusy has a width equal to `threads`. It may be included for single-threaded OCP interfaces.

12 SThreadBusy has a width equal to `threads`. It may be included for single-threaded OCP interfaces.

13 ControlBusy can only be included if both Control and ControlWr exist.

14 ControlWr can only be included if Control exists.

15 StatusBusy can only be included if Status exists.

16 StatusRd can only be included if Status exists.

Signal Directions

Figure 4 on page 29 summarizes all OCP signals. The direction of some signals (for example, MCmd) depends on whether the module acts as a master or slave, while the direction of other signals (for example, Control) depends on whether the module acts as a system or a core. The combination of these two choices provides four possible module configurations as shown in Table 13.

Table 13 Module Configuration Based on Signal Directions

	Acts as Core or has No System Signals	Acts as System
Acts as OCP Master	Master	System master
Acts as OCP Slave	Slave	System slave

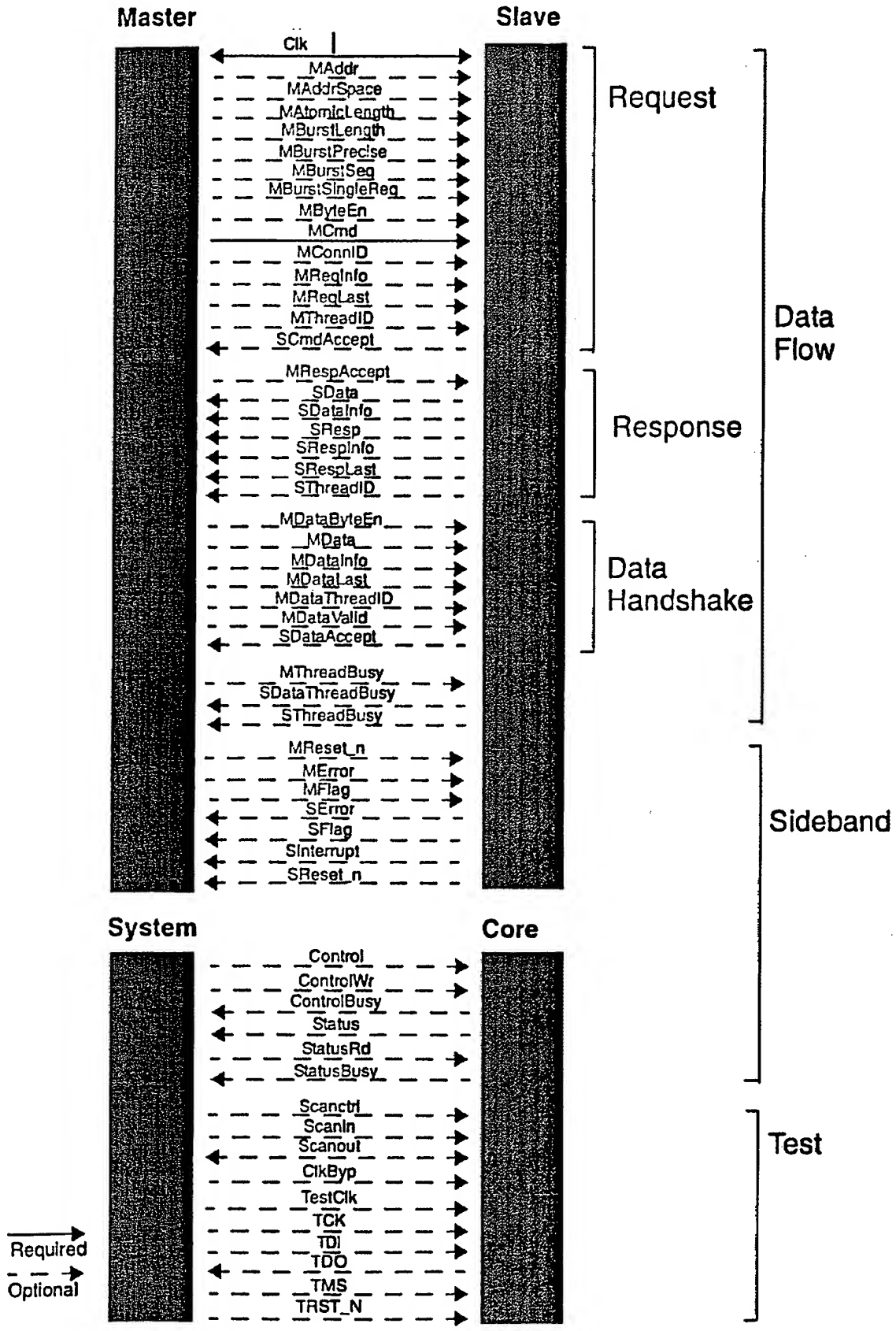
For example, if a module acts as OCP master and also as system, it is designated a system master. There is also a monitor type that observes all OCP signals. The permitted connectivity between interface types is shown in Table 14.

Table 14 Interface Types

Type	Connect To	Cannot Connect To
Master	System slave Slave	Master System master
Slave	System master Master	Slave System slave
System master	Slave System slave	Master System master
System slave	Master System master	Slave System slave
Monitor	Any except monitor	Monitor

The Clk signal is special in that it is always supplied by a third (external) entity that is neither of the two modules connected through the OCP interface.

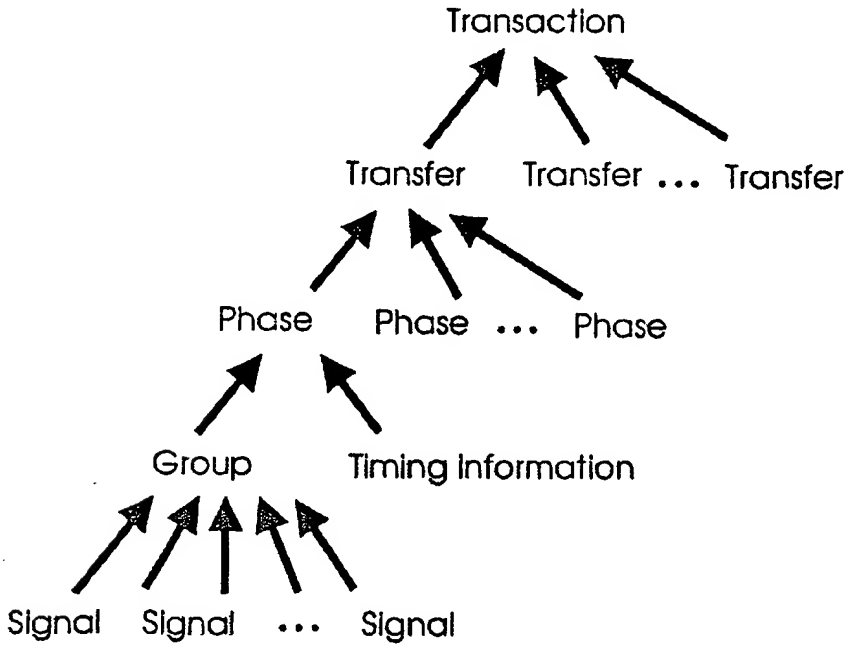
Figure 4 OCP Signal Summary



4 Protocol Semantics

This chapter defines the semantics of the OCP protocol by assigning meanings to the signal encodings described in the preceding chapter. Figure 5 provides a graphic view of the hierarchy of elements that compose the OCP.

Figure 5 Hierarchy of Elements



Signal Groups

Some OCP fields are grouped together because they must be active at the same time. The data flow signals are divided into three signal groups: request signals, response signals, and datahandshake signals. A list of the signals that belong to each group is shown in Table 15.

Table 15 OCP Signal Groups

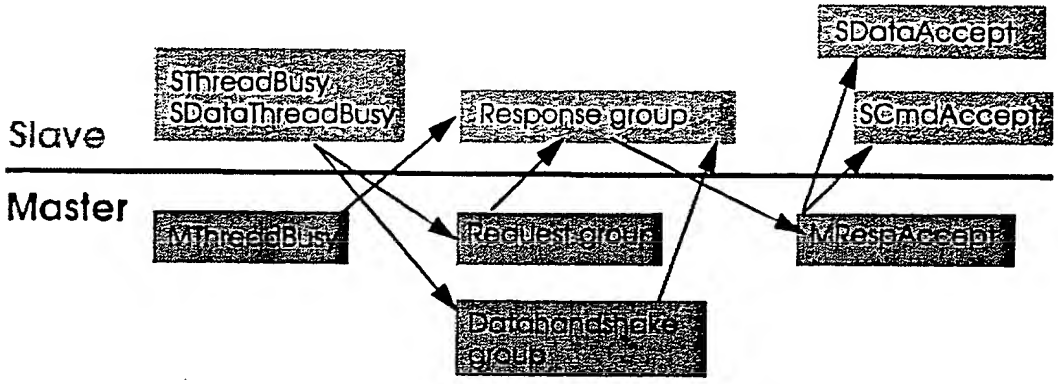
Group	Signal	Condition
Request	MAddr	always
	MAddrSpace	always
	MAtomicLength	always
	MBurstLength	always
	MBurstPrecise	always
	MBurstSeq	always
	MBurstSingleReq	always
	MByteEn	always
	MCmd	always
	MConnID	always
	MData*	datahandshake = 0
	MDataInfo*	datahandshake = 0
	MReqInfo	always
	MReqLast	always
	MThreadID	always
Response	SData	always
	SDataInfo	always
	SResp	always
	SRespInfo	always
	SRespLast	always
	SThreadID	always
Datahandshake	MData*	datahandshake = 1
	MDataByteEn	always
	MDataInfo*	datahandshake = 1
	MDataLast	always
	MDataThreadID	always
	MDataValid	always

* MData and MDataInfo belong to the request group, unless the `datahandshake` configuration parameter is enabled. In that case they belong to the datahandshake group.

Combinational Dependencies

It is legal for some signal or signal group outputs to be derived from inputs without an intervening latch point, that is combinationally. To avoid combinational loops, other outputs cannot be derived in this manner. Figure 6 describes a partial order of combinational dependency. For any arrow shown, the signal or signal group can be derived combinationally from the signal at the point of origin of the arrow or another signal earlier in the dependency chain. No other combinational dependencies are allowed.

Figure 6 Legal Combinational Dependencies Between Signals and Signal Groups



Combinational paths are not allowed within the sideband and test signals, or between those signals and the data flow signals. The only legal combinational dependencies are within the data flow signals. Data flow signals, however, may be combinationally derived from MReset_n and SReset_n.

For timing purposes, some of the allowed combinational paths are designated as preferred paths and are described in Table 29 on page 163.

Signal Timing and Protocol Phases

This section specifies when a signal can or must be valid.

Dataflow Signals

Signals in a signal group must all be valid at the same time.

- The request group is valid whenever a command other than Idle is presented on the MCmd field.
- The response group is valid whenever a response other than Null is presented on the SResp field.
- The datahandshake group is valid whenever a 1 is presented on the MDataValid field.

The accept signal associated with a signal group is valid only when that group is valid.

- The SCmdAccept signal is valid whenever a command other than Idle is presented on the MCmd field.
- The MRespAccept signal is valid whenever a response other than Null is presented on the SResp field.
- The SDataAccept signal is valid whenever a 1 is presented on the MDataValid field.

The signal groups map on a one-to-one basis to protocol phases. All signals in the group must be held steady from the beginning of a protocol phase until the end of that phase. Outside of a protocol phase, all signals in the corresponding group (except for the signal that defines the beginning of the phase) are "don't care".

In addition, the MData and MDataInfo fields are a "don't care" during read-type requests, and the SData and SDataInfo fields are a "don't care" for responses to write-type requests. Non-enabled data bytes in MData and bits in MDataInfo as well as non-enabled bytes in SData and bits in SDataInfo are a "don't care". The MDataByteEn field is a "don't care" during read-type transfers. If MDataByteEn is present, the MByteEn field is a "don't care" during write-type transfers.

- A request phase begins whenever the request group becomes active. It ends when the SCmdAccept signal is sampled by Clk as 1 during a request phase.
- A response phase begins whenever the response group becomes active. It ends when the MRespAccept signal is sampled by Clk as 1 during a response phase.

  If MRespAccept is not configured into the OCP interface (`respaccept = 0`) then MRespAccept is assumed to be on; that is the response phase is exactly one cycle long.
- A datahandshake phase begins whenever the datahandshake signal group becomes active. It ends when the SDataAccept signal is sampled by Clk as 1 during a datahandshake phase.

For all phases, it is legal to assert the corresponding accept signal in the cycle that the phase begins, allowing the phase to complete in a single cycle.

Phases in a Transfer

An OCP transfer consists of several phases as shown in Table 16. Every transfer has a request phase. Read-type requests always have a response phase. For write-type requests, the OCP can be configured with or without responses or datahandshake. Without a response, a write-type request completes upon completion of the request phase (posted write model).

Table 16 Phases In a Transfer for MBurstSinqleReq set to 0

MCmd	Phases	Condition
Read, ReadEx, ReadLinked	Request, response	always
Write, Broadcast	Request	datahandshake = 0 writeresp_enable = 0
Write, WriteNonPost, WriteConditional, Broadcast	Request, response	datahandshake = 0 writeresp_enable = 1
Write, Broadcast	Request, datahandshake	datahandshake = 1 writeresp_enable = 0
Write, WriteNonPost, WriteConditional, Broadcast	Request, datahandshake, response	datahandshake = 1 writeresp_enable = 1

Single request / multiple data bursts, described in "Single Request / Multiple Data Bursts (Packets)" on page 45, have a single request phase and multiple data transfer phases as shown in Table 17.

Table 17 Phases In a Transfer for MBurstSinqleReq set to 1

MCmd	Phases	Condition
Read	Request, L* response	always
Write, Broadcast	Request, L* datahandshake	datahandshake = 1 writeresp_enable = 0
Write, WriteNonPost, Broadcast	Request, L* datahandshake, response	datahandshake = 1 writeresp_enable = 1

*L refers to the burst length

Phase Ordering Within a Transfer

The OCP is causal: within each transfer a request phase must precede the associated datahandshake phase which in turn, must precede the associated response phase. The specific constraints are:

- A datahandshake phase cannot begin before the associated request phase begins, but can begin in the same Clk cycle.
- A datahandshake phase cannot end before the associated request phase ends, but can end in the same Clk cycle.
- A response phase cannot begin before the associated request phase begins, but can begin in the same Clk cycle.
- A response phase cannot end before the associated request phase ends, but can end in the same Clk cycle.

- If there is a datahandshake phase and a response phase, the response phase cannot begin before the associated datahandshake phase (or last datahandshake phase for single request/multiple data bursts) begins, but can begin in the same Clk cycle.
- If there is a datahandshake phase and a response phase, the response phase cannot end before the associated datahandshake phase (or last datahandshake phase for single request/multiple data bursts) ends, but can end in the same Clk cycle.

Phase Ordering Between Transfers

The ordering of transfers is determined by the ordering of their request phases.

- Since two phases of the same type belonging to different transfers both use the same signal wires, the phase of a subsequent transfer cannot begin before the phase of the previous transfer has ended. If the first phase ends in cycle x, the second phase can begin in cycle x + 1.
- The ordering of datahandshake phases must follow the order set by the request phases including multiple datahandshake phases for single request / multiple data bursts.
- The ordering of response phases must follow the order set by the request phases including multiple response phases for single request / multiple data bursts.
- For single request / multiple data bursts, a request or response phase is shared between multiple transfers. Each individual transfer must obey the ordering rules described in "Phase Ordering Within a Transfer" on page 35, even when a phase is shared with another transfer.
- Where no phase ordering is specified, by the previous rules, the effect of two transfers that are addressing the same location (as specified by MAddr, MAddrSpace, and MByteEn) must be the same as if the two transfers were executed in the same order but without any phase overlap. This ensures that read/write hazards yield predictable results.

  For example, on an OCP interface with datahandshake enabled, a read following a write to the same location cannot start its response phase until the write has started its datahandshake phase, otherwise the latest write data cannot be returned for the read.

Ungrouped Signals

Signals not covered in the description of signal groups and phases are MThreadBusy, SDataThreadBusy, and SThreadBusy. Without further protocol restriction, the cycle timing of the transition of each bit that makes up each of these three fields is not specified relative to the other dataflow signals. This means that there is no specific time for an OCP master or slave to drive these signals, nor a specific time for the signals to have the desired flow-control effect. It follows that without further restriction, MThreadBusy, SDataThreadBusy, and SThreadBusy can only be treated as a hint. It is

possible to force stricter semantics using the protocol configuration parameters mthreadbusy_exact, sdatathreadbusy_exact, and sthreadbusy_exact in the following way:

- If mthreadbusy_exact is enabled for a master and the master is unable to accept a response on a thread in a given cycle, it must set the MThreadBusy bit for that thread to 1 in that cycle. If a response is presented on a thread for which MThreadBusy is not set to 1 in the current cycle, it must be accepted by the master in that cycle.

  If mthreadbusy_exact is enabled for a slave, the slave must take the MThreadBusy signal into account for the thread selection of the current cycle, that is, it must not present a response on a thread for which the corresponding MThreadBusy bit is set to 1 in the cycle.

- If sdatathreadbusy_exact is enabled for a slave that is unable to accept a datahandshake phase on a thread in a given cycle, it must set the SDataThreadBusy bit for that thread to 1 in that cycle. If a datahandshake phase is presented on a thread for which SDataThreadBusy is not set to 1 in the current cycle, it must be accepted by the slave in that cycle.

  If sdatathreadbusy_exact is enabled for a master, the master must take the SDataThreadBusy signal into account for the thread selection of the current cycle, that is, it must not present a datahandshake phase on a thread for which the corresponding SDataThreadBusy bit is set to 1 in this cycle.

- If sthreadbusy_exact is enabled for a slave that is unable to accept a command on a thread in a given cycle, it must set the SThreadBusy bit for that thread to 1 in that cycle. If a command is presented on a thread for which SThreadBusy is not set to 1 in the current cycle, it must be accepted by the slave in that cycle.

  If sthreadbusy_exact is enabled for a master, the master must take the SThreadBusy signal into account for the thread selection of the current cycle, that is, it must not present a command on a thread for which the corresponding SThreadBusy bit is set to 1 in this cycle.

A multi-threaded OCP interface is guaranteed to be non-blocking only if all of the following conditions are satisfied:

1. cmdaccept is set to 0 and SCmdAccept is tied off to a value of 1.
2. If sthreadbusy is set to 1, sthreadbusy_exact is set to 1 for both master and slave.
3. If datahandshake is set to 1, dataaccept is set to 0 and SDataAccept is tied off to a value of 1.
4. If datahandshake is set to 1 and sdatathreadbusy is set to 1, sdatathreadbusy_exact is set to 1 for both master and slave.
5. respaccept is set to 0 and MRespAccept is tied off to a value of 1.
6. If mthreadbusy is set to 1, mthreadbusy_exact is set to 1 for both master and slave.

Sideband and Test Signals

Reset

The OCP interface provides an interface reset signal for each master and slave. At least one of these signals must be present. If both signals are present, the composite reset state of the interface is derived as the logical AND of the two signals (that is, the interface is in reset as long as one of the two resets is asserted).

Once one or both reset signals are sampled asserted by the rising edge of Clk, all incomplete transactions, transfers and phases are terminated and both master and slave must transition to a state where there are no pending OCP requests or responses. MReset_n and SReset_n must be asserted for at least 16 cycles of Clk to ensure that the master and slave reach a consistent internal state. When one or both of the reset signals are asserted in a given cycle, all other OCP signals must be ignored in that cycle. The master and slave must each be able to reach their reset state regardless of the values presented on the OCP signals. If the master or slave require more than 16 cycles of reset assertion, the requirement must be documented in the IP core specifications.

At the clock edge that all reset signals present are sampled deasserted, all OCP interface signals must be valid. In particular, it is legal for the master to begin its first request phase in the same clock cycle that reset is deasserted.

Interrupt, Error, and Core Flags

There is no specific timing associated with SInterrupt, SError, MFlag, MError, and SFlag. The timing of these signals is core-specific.

Status and Control

The following rules assure that control and status information can be exchanged across the OCP without any combinational paths from inputs to outputs and at the pace of a slow core.

- Control must be held steady for a full cycle after the cycle in which it has transitioned, which means it cannot transition more frequently than every other cycle. If ControlBusy was sampled active at the end of the previous cycle, Control must not transition in the current cycle. In addition, Control must be held steady for the first two cycles after reset is deasserted.
- If Control transitions in a cycle, ControlWr (if present) must be driven active for that cycle. ControlWr following the rules for Control, cannot be asserted in two consecutive cycles.
- ControlBusy allows a core to force the system to hold Control steady. ControlBusy may only start to be asserted immediately after reset, or in the cycle after ControlWr is asserted, but can be left asserted for any number of cycles.

- While StatusBusy is active, Status is a "don't care". StatusBusy enables a core to prevent the system from reading the current status information. While StatusBusy is active the core may not read Status. StatusBusy can be asserted at any time and be left asserted for any number of cycles.
- StatusRd is active for a single cycle every time the status register is read by the system. If StatusRd was asserted in the previous cycle, it must not be asserted in the current cycle, so it cannot transition more frequently than every other cycle.

Test Signals

Scanin and Scanout are "don't care" while Scanctrl is inactive (but the encoding of inactive for Scanctrl is core-specific).

TestClk is "don't care" while `ClkByp` is 0.

The timing of TRST_N, TCK, TMS, TDI, and TDO is specified in the IEEE 1149 standard.

Transfer Effects

A successful transfer is one that completes without error. For write-type requests without responses, there can be no in-band error indication. For all other requests, a non-ERR response (that is, a DVA or FAIL response) indicates a successful transfer. The FAIL response is legal only for WriteConditional commands. This section defines the effect that a successful transfer has on a slave. The request acts on the addressed location, where the term address refers to the combination of MAddr, MAddrSpace, and MByteEnable (or MDataByteEn, if applicable). Two addresses are said to match if they are identical in all components. Two addresses are said to conflict, if the mutual exclusion (lock or monitor) logic is built to alias the two addresses into the same mutual exclusion unit. The transfer effects of each command are:

Idle
: None.

Read
: Returns the latest value of the addressed location on the `SData` field.

ReadEx
: Returns the latest value of the addressed location on the SData field. Sets a lock for the initiating thread on that location. The next request on the thread that issued a ReadEx must be a Write or WriteNonPost to the matching address. Requests from other threads to a conflicting address that is locked are blocked from proceeding until the lock is unset. If the ReadEx request returns an ERR response, it is slave-specific whether the lock is actually set or not.

ReadLinked
: Returns the latest value of the addressed location on the SData field. Sets a reservation tag in a monitor for the corresponding thread on at least that

location. Requests of any type from any thread to a conflicting address that is reserved are not blocked from proceeding, but may clear the reservation tag.

Write/WriteNonPost
: Places the value on the MData field in the addressed location. Unlocks access to the matched address if locked by a ReadEx. Clears the reservation tags on any conflicting addresses set by other threads.

WriteConditional
: If a reservation tag is set for the matching address and for the corresponding thread, the write is performed as it would be for a Write or WriteNonPost. Simultaneously, the reservation tag is cleared for all threads on any conflicting address. If no reservation tag is set for the corresponding thread, the write is not performed, a FAIL response is returned, and no reservation tags are cleared.

Broadcast
: Places the value on the MData field in the addressed location that may map to more than one slave in a system-dependent way. Broadcast clears the reservation tags on any conflicting addresses set by other threads.

If a transfer is unsuccessful, the effect of the transfer is unspecified. It is up to higher-level protocols to determine what happened and handle any clean-up.

The synchronization commands ReadEx / Write, ReadEx / WriteNonPost, and ReadLinked / WriteConditional have special restrictions with regard to data width conversion and partial words. In systems where these commands are sent through a bridge or interconnect that performs wide-to-narrow data width conversion between two OCP interfaces, the initiator must issue only commands within the subset of partial words that can be expressed as a single word of the narrow OCP interface. For maximum portability, single-byte synchronization operations are recommended.

Partial Word Transfers

An OCP interface may be configured to include partial word transfers by using either the MByteEn field, or the MDataByteEn field, or both.

- If neither field is present, then only whole word transfers are possible.
- If only MByteEn is present, then the partial word is specified by this field for both read type transfers and write type transfers as part of the request phase.
- If only MDataByteEn is present, then the partial word is specified by this field for write type transfers as part of the datahandshake phase, and partial word reads are not supported.
- If both MByteEn and MDataByteEn are present, then MByteEn specifies partial words for read transfers as part of the request phase, and MDataByteEn specifies partial words for write transfers as part of the datahandshake phase.

It is legal to use a request with all byte enables deasserted. Such requests must follow all the protocol rules, except that they are treated as no-ops by the slave: the response phase signals SData and SDataInfo are "don't care" for read-type commands, and nothing is written for write-type commands.

Posting Semantics

The OCP includes a basic Write command. Typically, a system designer decides what write completion model to assign to a core's write commands. If the OCP is configured to not respond to write-type commands (`writeresp_enable` set to 0), a posted write completion model is assumed. It is also possible to achieve a non-posted model by not accepting the command until the write completes, but this de-pipelines the OCP interface and is not recommended.

If the OCP is configured to respond to write-type commands (`writeresp_enable` set to 1), either a posted or a non-posted write completion model can be used. For cores that need to determine on a per-transfer basis whether a write is to be treated as posted or non-posted, the OCP provides the WriteNonPost command.

Endianness

An OCP interface by itself is inherently endian-neutral. Data widths must match between master and slave, addressing is on an OCP word granularity, and byte enables are tied to byte lanes (data bits) without tying the byte lanes to specific byte addresses.

The issue of endianness arises in the context of multiple OCP interfaces, where the data widths of the initiator of a request and the final target of that request do not match. Examples are a bridge or a more general interconnect used to connect OCP-based cores.

When the OCP interfaces differ in data width, the interconnect must associate an endianness with each transfer. It does so by associating byte lanes and byte enables of the wider OCP with least-significant word address bits of the narrower OCP. Packing rules, described in "Packing" on page 43 must also be obeyed for bursts.

OCP interfaces can be designated as little, big, both, or neutral with respect to endianness. This is specified using the protocol parameter `endian` described in "Endianness" on page 49. A core that is designated as both typically represents a device that can change endianness based upon either an internal configuration register or an external input. A core that is designated as neutral typically represents a device that has no inherent endianness. This indicates that either the association of an endianness is arbitrary (as with a memory, which traditionally has no inherent endianness) or that the device only works with full-word quantities (when `byteen` and `mdatabyteen` are set to 0).

When all cores have the same endianness, an interconnect should match the endianness of the attached cores. The details of any conversion between cores of different endianness is implementation-specific.

Burst Definition

A *burst* is a set of transfers that are linked together into a transaction having a defined address sequence and number of transfers. There are three general categories of bursts:

Imprecise bursts
: Request information is given for each transfer. Length information may change during the burst.

Precise bursts
: Request information is given for each transfer, but length information is constant throughout the burst.

Single request / multiple data bursts (also known as packets)
: Also a precise burst, but request information is given only once for the entire burst.

To express bursts on the OCP interface, at least the address sequence and length of the burst must be communicated, either directly using the MBurstSeq and MBurstLength signals, or indirectly through an explicit constant tie-off as described in "Signal Mismatch Tie-off Rules" on page 53.

A single (non-burst) request on an OCP interface with burst support is encoded as a request with any legal burst address sequence and a burst length of 1.

The ReadEx, ReadLinked, and WriteConditional commands can not be used as part of a burst. The unlocking Write or WriteNonPost command associated with a ReadEx command also can not be used as part of a burst.

Burst Address Sequence

The relationship of the MBurstSeq encodings and corresponding address sequences are shown in Table 18. The table also indicates whether a burst sequence type is packing or not, a concept discussed on page 43.

Table 18 Burst Address Sequences

Mnemonic	Name	Address Sequence	Packing
DFLT1	custom (packed)	user-specified	yes
DFLT2	custom (not packed)	user-specified	no
INCR	incrementing	incremented by OCP word size each transfer*	yes
STRM	streaming	constant each transfer	no

Mnemonic	Name	Address Sequence	Packing
UNKN	unknown	none specified	implementation specific
WRAP	wrapping	like INCR, except wrap at address boundary aligned with MBurstLength * OCP word size	yes
XOR	exclusive OR	see below for precise definition	yes

*Bursts must not wrap around the OCP address size.

The address sequence for exclusive OR bursts is as follows. Let BASE be the lowest byte address in the burst, which must be aligned with the total burst size. Let FIRST_OFFSET be the byte offset (from BASE) of the first transfer in the burst. Let CURRENT_COUNT be the count of the current transfer in the burst, starting at 0. Let WORD_SHIFT be the log2 of the OCP word size in bytes. Then the current address of the transfer is BASE | (FIRST_OFFSET ^ (CURRENT_COUNT << WORD_SHIFT)).

Burst address sequence UNKN is used when the address sequence is not statically known for the burst. In contrast, DFLT1 and DFLT2 address sequences are known, but are core or system specific.

Burst address sequences WRAP and XOR can only be used for precise bursts with a power-of-two burst length.

Not all masters and slaves need to support all burst sequences. A separate protocol parameter described in "Optional Burst Sequences" on page 47 is provided for each burst sequence to indicate support for that burst sequence.

Byte Enable Restrictions

Burst address sequences STRM and DFLT2 must have at least one byte enable asserted for each transfer in the burst. Bursts with the STRM address sequence must have the same byte enable pattern for each transfer in the burst.

Packing

Packing allows the system to make use of the burst attributes to improve the overall data transfer efficiency in the face of multiple OCP interfaces of different data widths. For example, if a bridge is translating a narrow OCP to a wide OCP, it can aggregate (or pack) the incoming narrow transfers into a smaller number of outgoing wide transfers. Burst address sequences are classified as either packing, or not packing.

For burst address sequences that are packing, the conversion between different OCP data widths is achieved through aggregation or splitting. Narrow OCP words are collected together to form a wide OCP word. A wide OCP word is split into several narrow OCP words. The byte-specific portion of MDataInfo and SDataInfo is aggregated or split with the data. The transfer-specific portion of MDataInfo and SDataInfo is unaffected. The packing and unpacking order depends on endianness as described on page 41.

For burst address sequences that are not packing, conversion between different OCP data widths is achieved using padding and stripping. A narrow OCP word is padded to form a wide OCP word with only the relevant byte enables turned on. A wide OCP word is stripped to form a narrow OCP word. The byte-specific portion of MDataInfo and SDataInfo is zero-padded or stripped with the data. The transfer-specific portion of MDataInfo and SDataInfo is unaffected. Width conversion can be performed reliably only if the wide OCP interface has byte enables associated with it. For wide to narrow conversion the byte enables are restricted to a subset that can be expressed within a single word of the narrow OCP interface.

Since the address sequence of DFLT1 is user-specified, the behavior of DFLT1 bursts through data width conversion is implementation-specific.

Burst Length, Precise and Imprecise Bursts

The MBurstLength field indicates the number of transfers in the burst.

Precise bursts (MBurstPrecise set to 1)
: MBurstLength must be held constant throughout the burst, so the exact burst length can be obtained from the first transfer. A precise burst is completed by the transfer of the correct number of OCP words. Precise bursts are recommended over imprecise bursts because they allow for increased hardware optimization.

Imprecise bursts (MBurstPrecise set to 0)
: MBurstLength can change throughout the burst, and indicates the current best guess of the number of transfers left in the burst (including the current one). An imprecise burst is completed by an MBurstLength of 1.

Constant Fields in Bursts

MCmd, MAddrSpace, MConnID, MBurstPrecise, MBurstSingleReq, MBurstSeq, MAtomicLength, and MReqInfo must all be held steady by the master for every transfer in a burst, regardless of whether the burst is precise or imprecise. Similarly, SRespInfo must be held steady by the slave for every transfer in a burst.

Atomicity

When interleaving requests from different initiators on the way to or at the target, the master uses MAtomicLength to indicate the number of OCP words within a burst that must be kept together as an atomic quantity. If MAtomicLength is greater than the actual length of the burst, the atomicity requirement ends with the end of the burst. Specifying atomicity requirements explicitly is especially useful when multiple OCP interfaces are involved that have different data widths.

For master cores, it is best to make the atomic size as small as required and, if possible, to keep the groups of atomic words address-aligned with the group size.

Single Request / Multiple Data Bursts (Packets)

MBurstSingleReq specifies whether a burst can be communicated using a single request / multiple data protocol. When MBurstSingleReq is 0, each request has a single data word associated with it. When MBurstSingleReq is 1, each request may have multiple data words associated with it, according to the value of MBurstLength. MBurstSingleReq may be set to 1 only if MBurstPrecise is set to 1. In addition, if any write-type commands are enabled, `datahandshake` must be set to 1.

When MBurstSingleReq is set to 1, write type transfers have MBurstLength phases and a single response phase (if `writeresp_enable=1`) per request; while read-type transfers have MBurstLength response phases per request as shown in Table 17 on page 35.

For write type transfers when MBurstSingleReq is set to 1 and the MDataByteEn field is present, that field in each data transfer phase specifies the partial word pattern for the phase. When MBurstSingleReq is set to 1 and the MDataByteEn field is not present, the MByteEn pattern of the request phase applies to all data transfer phases.

For read type transfers when MBurstSingleReq is set to 1, the MByteEn field specifies the byte enable pattern that is applied to all data transfers in the burst.

MReqLast, MDataLast, SRespLast

Optional signals MReqLast, MDataLast, and SRespLast provide redundant information that indicates the last request, datahandshake, and response phase in a burst, respectively. These signals are provided as a convenience to the recipient of the signal. To avoid separate counting mechanisms to track bursts, cores that have the information available internally are encouraged to provide it at the OCP interface.

MReqLast is 0 for all request phases in a burst except the last one. MReqLast is 1 for the last request phase in a burst, for single request / multiple data bursts, and for single requests.

MDataLast is 0 for all datahandshake phases in a burst except the last one. MDataLast is 1 for the last datahandshake phase in a burst and for the only datahandshake phase of a single request.

SRespLast is 0 for all response phases in a burst except the last one. SRespLast is 1 for the last response phase in a burst, for the response to a write-type single request / multiple data burst, and for the response to a single request.

Threads and Connections

All transfers within a single thread must remain ordered. (See "Phase Ordering Between Transfers" on page 36.)

Using multiple threads, it is possible to support concurrent activity, and out-of-order completion of transfers. All transfers within a given thread must remain strictly ordered, but there are no ordering rules for transfers that are in different threads. Mapping of individual requests and responses to threads is handled through the MThreadID and SThreadID fields respectively. If datahandshake has been enabled when multiple threads are present, there must also be an MDataThreadID field to annotate the datahandshake phase. If `datahandshake` is set to 1 and `sdatathreadbusy` is set to 0, the order of datahandshake phases must follow the order of request phases across all threads. If `sdatathreadbusy` is set to 1, the request order and datahandshake order are independent across threads.

The use of thread IDs allows two entities that are communicating over an OCP interface to assign transfers to particular threads. If one of the communicating entities is itself a bridge to another OCP interface, the information about which transfers are part of which thread must be maintained by the bridge, but the actual assignment of thread IDs is done on a per-OCP-interface basis. There is no way for a slave on the far side of a bridge to extract the original thread ID unless the slave design comprehends the characteristics of the bridge.

Use connections whenever source thread information about a request must be sent end-to-end from master to slave. Any bridges in the path between the end-to-end partners preserve the connection ID, even as thread IDs are reassigned on each OCP interface in the path. The MConnID field transfers the connection ID during the request phase. Since this establishes the mapping onto a thread ID, the other phases do not require a connection ID but are unambiguous with only a thread ID.

The SThreadBusy, SDataThreadbusy, and MThreadBusy signals are used to indicate that a particular thread is busy. The protocol parameters `sthreadbusy_exact`, `sdatathreadbusy_exact`, and `mthreadbusy_exact` can be used to force precise semantics for these signals and assure that a multi-threaded OCP interface never blocks. For more information, see "Ungrouped Signals" on page 36.

OCP Configuration

This section describes configuration options that control interface capabilities.

Protocol Options

Optional Commands

Not all devices support all commands. Each command in Table 19 has an enabling parameter to indicate if that command is supported.

Table 19 Command Enabling Parameters

Command	Parameter
Broadcast	broadcast_enable
Read	read_enable
ReadEx	readex_enable
ReadLinked and WriteConditional	rdlwrc_enable
Write	write_enable
WriteNonPost	writenonpost_enable

The following conditions apply to command support:

- A master with one of these options set to 0 must not generate the corresponding command.
- A slave with one of these options set to 0 cannot service the corresponding command.
- At least one of the command enables must be set to 1.
- `writenonpost_enable` can be set to 1 only if `writeresp_enable` is set to 1.
- `rdlwrc_enable` can be set to 1 only if `writeresp_enable` is set to 1.
- If any read-type command is enabled or `writeresp_enable` is set to 1, `resp` must be set to 1.
- If `readex_enable` is set to 1, `write_enable` or `writenonpost_enable` must be set to 1.

Optional Burst Sequences

Not all masters and slaves need to support all burst address sequences. Table 20 lists the protocol parameter for each burst sequence. A master that has the parameter set to 1 may generate the corresponding burst sequence. A slave

that has the parameter set to 1 can service the corresponding burst sequence. If MBurstLength is not tied off to a value of 1, at least one of the burst sequence parameters must be enabled.

Table 20 Burst Sequence Parameters

Burst Sequence	Parameter
DFLT1	burstseq_dflt1_enable
DFLT2	burstseq_dflt2_enable
INCR	burstseq_incr_enable
STRM	burstseq_strm_enable
UNKN	burstseq_unkn_enable
WRAP	burstseq_wrap_enable
XOR	burstseq_xor_enable

For additional information describing bursts, see "Burst Definition" on page 42.

Byte Enable Patterns

Not all devices support all allowable byte enable patterns. The `force_aligned` parameter limits byte enable patterns on MByteEn and MDataByteEn to be power-of-two in size and aligned to that size. The byte enable pattern of all 0s is explicitly included in the legal `force_aligned` patterns.

- A master with this option set to 1 must not generate any byte enable patterns that are not force aligned.
- A slave with this option set to 1 cannot handle any byte enable patterns that are not force aligned.

`force_aligned` can be set to 1 only when `data_wdth` is set to a power-of-two value.

Burst Alignment

The `burst_aligned` parameter provides information about the length and alignment of INCR bursts issued by a master and can be used to optimize the system. Setting `burst_aligned` to 1 requires all INCR bursts to:

- Have an exact power-of-two number of transfers
- Have their starting address aligned with their total burst size
- Be issued as precise bursts

Exact ThreadBusy

The sthreadbusy_exact and mthreadbusy_exact parameters require strict semantics for the SThreadBusy and MThreadBusy signals to achieve a guarantee that a multi-threaded OCP interface never blocks. See "Ungrouped Signals" on page 36 for a precise definition of these parameters. The following conditions must be true:

- When sthreadbusy_exact is set to 1, cmdaccept must be set to 0, and SCmdAccept is tied off to a value of 1.
- When sthreadbusy_exact is set to 1, datahandshake is set to 1, and sdatathreadbusy is set to 0, dataaccept must be set to 0 and SDataAccept is tied off to a value of 1.
- When sdatathreadbusy_exact is set to 1, dataaccept must be set to 0 and SDataAccept is tied off to a value of 1.
- When mthreadbusy_exact is set to 1, respaccept must be set to 0, and MRespAccept is tied off to a value of 1.

The following conditions are illegal because no flow control can be exerted:

- sthreadbusy_exact is set to 0, cmdaccept is set to 0, but sthreadbusy is set to 1.
- mthreadbusy_exact is set to 0, respaccept is set to 0, but mthreadbusy is set to 1.

Endianness

The endian parameter specifies the endianness of a core. The behavior of each endianness choice is summarized in Table 21.

Table 21 Endianness

Endianness	Description
little	core is little-endian
big	core is big-endian
both	core can be either big or little endian, depending on its static or dynamic configuration (e.g. CPUs)
neutral	core has no inherent endianness (e.g. memories, cores that deal only in OCP words)

As far as OCP is concerned, little endian means that lower addresses are associated with lower numbered data bits (byte lanes), while big endian means that higher addresses are associated with lower numbered data bits (byte lanes). This becomes significant when packing is concerned (see "Packing" on page 43). In addition, for non-power-of-2 data widths, tie-off padding is always added at the most significant end of the OCP word. See "Endianness" on page 41 for additional information.

Phase Options

The datahandshake parameter allows write data to have a handshake interface separate from the request group.

Datahandshake

If datahandshake is set to 1, the MDataValid and SDataAccept signals are added to the OCP interface, a separate datahandshake phase is added, and the MData and MDataInfo fields are moved from the request group to the datahandshake group.

Request and Data Together

While datahandshake is required for OCP interfaces that are capable of communicating single request / multiple data bursts, a fully separated datahandshake may be overkill for some cores. The parameter reqdata_together is used to specify that the request and datahandshake phases of the first transfer in a single request / multiple data write-type burst begin and end together.

A master with reqdata_together set to 1 must present the request and first write data word in the same cycle and can expect that the slave will accept them together. If sthreadbusy_exact and sdatathreadbusy_exact are both set to 1, this implies that a request and first write data can be presented only when both SThreadBusy and SDataThreadBusy for the corresponding thread are 0. A slave with reqdata_together set to 1 must accept the request and first write data word in the same cycle and can expect that they will be presented together.

The parameter reqdata_together can only be set to 1 if burstsinglereq is set to 1, or burstsinglereq is set to 0 and MBurstSingleReq is tied off to 1.

Write Responses

To configure the OCP to include response phases for write-type requests, use the writeresp_enable parameter.

- If responses are not enabled on writes (writeresp_enable set to 0), then all write commands complete on command acceptance, and the WriteNonPost, WriteConditional, and ReadLinked commands are not allowed.
- If responses are enabled (writeresp_enable set to 1), writes may follow either a posted or non-posted model depending on when the response is returned. For this case, resp must be set to 1.

Signal Options

The configuration parameters described in "Signal Configuration" on page 25, not only configure the corresponding signal into the OCP interface, but also enable the function. For example, if the burstseq and burstlength parameters are enabled the MBurstSeq and MBurstLength fields are added and the interface also supports burst extensions as described in "Burst Definition" on page 42.

Minimum Implementation

A minimal OCP implementation must support at least the basic OCP dataflow signals. OCP-interoperable masters and slaves must support the command type Idle and at least one other command type.

If the SResp field is present in the OCP interface, OCP-interoperable masters and slaves must support response types NULL and DVA. The ERR response type is optional and should only be included if the OCP-interoperable slave has the ability to report errors. All OCP masters must be able to accept the ERR response. If `rdlwrc_enable` is set to 1, the FAIL response type must be supported by OCP masters and slaves.

OCP Interface Interoperability

Two devices connected together each have their own OCP configuration. The two interfaces are only interoperable (allowing the two devices to be connected together and communicate using the OCP protocol semantics) if they are interoperable at the core, protocol, phase, and signal levels.

1. At the core level:
   - One interface must act as master and the other as slave.
   - If system signals are present, one interface must act as core and the other as system.
2. At the protocol level, the following conditions determine interface interoperability:
   - If the slave has `read_enable` set to 0, the master must have `read_enable` set to 0, or it must not issue Read commands.
   - If the slave has `readex_enable` set to 0, the master must have `readex_enable` set to 0, or it must not issue ReadEx commands.
   - If the slave has `rdlwrc_enable` set to 0, the master must have `rdlwrc_enable` set to 0, or it must not issue either ReadLinked or WriteConditional commands.
   - If the slave has `write_enable` set to 0, the master must have `write_enable` set to 0, or it must not issue Write commands.
   - If the slave has `writenonpost_enable` set to 0, the master must have `writenonpost_enable` set to 0, or it must not issue WriteNonPost commands.
   - If the slave has `broadcast_enable` set to 0, the master must have `broadcast_enable` set to 0, or it must not issue Broadcast commands.
   - If the slave has `burstseq_incr_enable` set to 0, the master must have `burstseq_incr_enable` set to 0, or it must not issue INCR bursts.

- If the slave has burstseq_strm_enable set to 0, the master must have burstseq_strm_enable set to 0, or it must not issue STRM bursts.
- If the slave has burstseq_dflt1_enable set to 0, the master must have burstseq_dflt1_enable set to 0, or it must not issue DFLT1 bursts.
- If the slave has burstseq_dflt2_enable set to 0, the master must have burstseq_dflt2_enable set to 0, or it must not issue DFLT2 bursts.
- If the slave has burstseq_wrap_enable set to 0, the master must have burstseq_wrap_enable set to 0, or it must not issue WRAP bursts.
- If the slave has burstseq_xor_enable set to 0, the master must have burstseq_xor_enable set to 0, or it must not issue XOR bursts.
- If the slave has burstseq_unkn_enable set to 0, the master must have burstseq_unkn_enable set to 0, or it must not issue UNKN bursts.
- If the slave has force_aligned, the master has force_aligned or it must limit itself to aligned byte enable patterns.
- Configuration of the mdatabyteen parameter is identical between master and slave.
- If the slave has burst_aligned, the master has burst_aligned or it must limit itself to issue all INCR burst using burst_aligned rules.
- If the interface includes SThreadBusy, the sthreadbusy_exact parameter is identical between master and slave.
- If the interface includes MThreadBusy, the mthreadbusy_exact parameter is identical between master and slave.
- If the interface includes SDataThreadBusy, the sdatathreadbusy_exact parameter is identical between master and slave.
- All combinations of the endian parameter between master and slave are interoperable as far as the OCP interface is concerned. There may be core-specific issues if the endianness is mismatched.

3. At the phase level the two interfaces are interoperable if:
   - Configuration of the datahandshake parameter is identical between master and slave.
   - Configuration of the writeresp_enable parameter is identical between master and slave.
   - Configuration of the reqdata_together parameter is identical between master and slave.
4. At the signal level, two interfaces are interoperable if:

- data_wdth is identical for master and slave, or if one or both data_wdth configurations are not a power-of-two, if that data_wdth rounded up to the next power-of-two is identical for master and slave.
- The master and slave both have mreset or sreset set to 1.
- If the master has mreset set to 1, the slave has mreset set to 1.
- If the slave has sreset is set to 1, the master has sreset set to 1.
- The tie-off rules, described in the next section are observed for any mismatch at the signal level for fields other than MData and SData.

Signal Mismatch Tie-off Rules

Width mismatch for all fields other than MData and SData is handled through a set of signal tie-off rules. The rules state whether a master and slave that are mismatched in a particular field width configuration are interoperable, and if so how they must be connected together by tying off the mismatched signals.

If there is a width mismatch between master and slave for a particular signal configuration the following rules apply:

- If there are more outputs than inputs (the driver of the field has a wider configuration than the receiver of the field) the low-order output bits are connected to the input bits, and the high-order output bits are lost. The interfaces are interoperable if the sender of the field explicitly limits itself to encodings that only make use of the bits that are within the configuration of the receiver of the field.
- If there are more inputs than outputs (the driver of field has a narrower configuration than the receiver of the field) the low-order input bits are connected to the output bits, and the high-order input bits are tied to logical 0. The interfaces are always interoperable, but only a portion of the legal encodings are used on that field.

If one of the cores has a signal configured and the other does not, the following rules apply:

- If the core that would be the driver of the field does not have the field configured, the input is tied off to the constant specified in the driving core's configuration, or if no constant tie-off is specified, to the default tie-off constant (see Table 12 on page 25). The interfaces are interoperable if the encodings supported by the receiver's configuration of the field include the tie-off constant.
- If the core that would be the receiver of the field does not have the field configured, the output is lost. The receiver of the signal must behave as though in every phase it were receiving the tie-off constant specified in its configuration, or if no constant tie-off is specified, the default tie-off constant (see Table 12 on page 25). The interfaces are interoperable if the driver of the signal can limit itself to only driving the tie-off constant of the receiver.

If neither core has a signal configured, the interfaces are interoperable if both cores have the same tie-off constant, where the tie-off constant is either explicitly specified, or if no constant tie-off is specified explicitly, is the default tie-off (see Table 12 on page 25).

While the tie-off rules allow two mismatched cores to be connected to one another, this may not be enough to guarantee meaningful communication, especially when core-specific encodings are used for signals such as MReqInfo.

As the previous rules suggest, specifying core specific tie-off constants that are different than the default tie-offs for a signal (see Table 12 on page 25) makes it less likely that the core will be interoperable with other cores.

Configuration Parameter Defaults

To assure OCP interface interoperability between a master and a slave requires complete knowledge of the OCP interface configuration of both master and slave. This is achieved by a combination of (a) requiring some parameters to be explicitly specified for each core, and (b) defining defaults that are used when a parameter is not explicitly specified for a core.

Table 22 lists all configuration parameters. For parameters that do not need to be specified, a default value is listed, which is used whenever an explicit parameter value is not specified. Certain parameters are always required in certain configurations, and for these no default is specified.

Table 22 Configuration Parameter Defaults

Type	Parameter	Default
Protocol	broadcast_enable	0
	burst_aligned	0
	burstseq_dflt1_enable	0
	burstseq_dflt2_enable	0
	burstseq_incr_enable	1
	burstseq_strm_enable	0
	burstseq_unkn_enable	0
	burstseq_wrap_enable	0
	burstseq_xor_enable	0
	endian	little
	force_aligned	0
	mthreadbusy_exact	0
	rdlwrc_enable	0

Type	Parameter	Default
	read_enable	1
	readex_enable	0
	sdatathreadbusy_exact	0
	sthreadbusy_exact	0
	write_enable	1
	writenonpost_enable	0
Phase	datahandshake	0
	reqdata_together	0
	writeresp_enable	0
Signal (Dataflow)	addr	1
	addr_wdth	No default - must be explicitly specified if addr is set to 1
	addrspace	0
	addrspace_wdth	No default - must be explicitly specified if addrspace is set to 1
	atomiclength	0
	atomiclength_wdth	No default - must be explicitly specified if atomiclength is set to 1
	burstlength	0
	burstlength_wdth	No default - must be explicitly specified if burstlength is set to 1
	burstprecise	0
	burstseq	0
	burstsinglereq	0
	byteen	0
	cmdaccept	1
	connid	0
	connid_wdth	No default - must be explicitly specified if connid is set to 1
	dataaccept	0
	datalast	0
	data_wdth	No default - must be explicitly specified if mdata or sdata is set to 1
	mdata	1
	mdatabyteen	0
	mdatainfo	0

Type	Parameter	Default
Signal (Dataflow)	mdatainfo_wdth	No default - must be explicitly specified if mdatainfo is set to 1
	mdatainfobyte_wdth	
	mthreadbusy	0
	reqinfo	0
	reqinfo_wdth	No default - must be explicitly specified if reqinfo is set to 1
	reqlast	0
	resp	1
	respaccept	0
	respinfo	0
	respinfo_wdth	No default - must be explicitly specified if respinfo is set to 1
	resplast	0
	sdata	1
	sdatainfo	0
	sdatainfo_wdth	No default - must be explicitly specified if sdatainfo is set to 1
	sdatainfobyte_wdth	
	sdatathreadbusy	0
	sthreadbusy	0
	threads	1

Type	Parameter	Default
Signal (Sideband)	control	0
	controlbusy	0
	control_wdth	No default - must be explicitly specified if control is set to 1
	controlwr	0
	interrupt	0
	merror	0
	mflag	0
	mflag_wdth	No default - must be explicitly specified if mflag is set to 1
	mreset	No default - must be explicitly specified
	serror	0
	sflag	0
	sflag_wdth	No default - must be explicitly specified if sflag is set to 1
	sreset	No default - must be explicitly specified
	status	0
	statusbusy	0
	statusrd	0
	status_wdth	No default - must be explicitly specified if status is set to 1
Signal (Test)	clkctrl_enable	0
	jtag_enable	0
	jtagtrst_enable	0
	scanctrl_wdth	0
	scanport	0
	scanport_wdth	No default - must be explicitly specified if scanport is set to 1

5 Interface Configuration File

The interface configuration file describes a group of signals, called a bundle. For OCP interfaces, the bundle is pre-defined, and no interface configuration file is required. If you are using an interface other than OCP in your core RTL configuration file, the interface configuration file is required.

Name the file <bundle-name>_intfc.conf where bundle-name is the name given to the bundle that is being defined in the file.

Lexical Grammar

The lexical conventions used in the interface configuration file are:

<name> : (<letter> | '_') (<letter> | '_' | <digit>)*
<letter> : 'a' .. 'z' | 'A' .. 'Z'
<digit> : '0' .. '9'

<number> : <integer> | <float>

<integer> : <digit>+

<float> : <mantissa> [<exponent>]
<mantissa>: (<integer> '.') |('.' <integer>) |(<integer> '.' <integer>)
<exponent>: ('e' | 'E') ['+' | '-'] <integer>

Syntax

The interface configuration file is written using standard Tcl syntax. Syntax is described using the following conventions:

[ ] optional construct
| or, alternate constructs
* zero or more repetitions
+ one or more repetitions
< > enclose names of syntactic units
( ) are used for grouping
{ } are part of the format and are required. An open brace must always appear on the same line as the statement
comments

The syntax of the interface configuration file is:

```
version <version_string>
bundle <bundle_name> {<bundle_stmt>+}
```

where:

```
<bundle_stmt>:
        | bundle_version <version_string>
        | interface_types <interface_type-name>+
        | net <net_name> {<net_stmt>*}
        | proprietary <vendor_code> <organization_name>
          {<proprietary_statements>}
<net_stmt>:
        | direction (input|output|inout)+
        | width <number-of-bits>
        | vhdl_type <type-string>
        | type <net-type>
        | proprietary <vendor_code> <organization_name>
          {<proprietary_statements>}
```

The file must contain a single version statement followed by a single bundle statement. The bundle statement must contain exactly one interface_types statement, and one or more net statements. Each net statement must contain exactly one direction statement and may contain additional statements of other types.

version
: The version statement identifies the version of the interface configuration file format. The version string consists of major and minor version numbers separated by a decimal. The current version is 2.5.

bundle
: This statement is required and indicates that a bundle is being defined instead of a core or a chip. Make the bundle-name the same name as the one used in the file name.

bundle_version
: If there are multiple versions use the bundle_version statement to identify the version of the bundle. The version string consists of major and minor version numbers separated by a decimal. If no bundle_version statement is present, the default version 1.0 is used. The current bundle version of OCP is 2.0.

interface_types
: The interface_types statement lists the legal values for the interface types associated with the bundle. Interface types are used by the toolset in conjunction with the direction statement to determine whether an interface uses a net as an input or output signal. This statement is required and must have at least one type defined.

  Predefined interface types for OCP bundles are slave, master, system_slave, system_master, and monitor. These are explained in Table 14 on page 28.

net
: The net statement defines the signals that comprise the bundle. There should be one net statement for each signal that is part of the bundle. A net can also represent a bus of signals. In this case the net width is specified using the width statement. If no width statement is provided, the net width defaults to one. A bundle is required to contain at least one net. The net-name field is the same as the one used in the net-name field of the port statements in the core rtl file described in Chapter 6.

proprietary
: For a description, see "Proprietary Statement" on page 73.

direction
: The direction statement indicates whether the net is an input, output, or inout. This field is required and must have as many direction-values as there are interface types. The order of the values must duplicate the order of the interface types in the interface_types statement. The legal values are input, output, and inout.

vhdl_type
: By default VHDL signals and ports are assumed to be std_logic and std_logic_vector, but if you have ports on a core that are of a different type, the `vhdl_type` command can be used on a net. This type will be used only when soccomp is run with the `design_top=vhdl` option to produce a VHDL top-level netlist.

type
: The type statement specifies that a net has special handling needs for downstream tools such as synthesis and layout. Table 23 shows the allowed <net-type> options. If no <net-type> is specified, normal is assumed.

Table 23 net-type Options

<net-type>	Description
normal	default for nets without special handling needs
clock	clock net
reset	reset net
scan_in	scan input net
scan_out	scan output net
scan_enable	scan enable net, serves as mode control between functional and scan data inputs
test_mode	test mode net, puts logic into a special mode for use during production testing
jtag_tck	JTAG test clock
jtag_tdi	JTAG test data in
jtag_tdo	JTAG test data out
jtag_tms	JTAG test mode select
jtag_trstn	JTAG test logic reset

proprietary
For a description, see "Proprietary Statement" on page 73.

The following example defines an SRAM interface. The bundle being defined is called sram16.

```
bundle "sram16" {

   # Two interface types are defined, one is labeled
   # "controller" and the other is labeled "memory"
   interface_types controller memory

   # A net named Address is defined to be part of this bundle.
   net "Address" {

      # The direction of the "Address" net is defined to be
      # "output" for interfaces of type "controller" and "input"
      # for interfaces of type "memory".
      direction output input

      # The width statement indicates that there are 14 bits in
      # the "Address" net.
      width 14
   }
   net "WData" {
      direction output input
      width 16
   }
   net "RData" {
      # The direction of the "RData" net is defined to be
```

```
        # "input" for bundle of type "controller" and  "output" for
        # bundles of type "memory".
        direction input output
        width 16
    }
    net "We_n" {
        direction output input
    }
    net "Oe_n" {
        direction output input
    }
    net "Reset" {
        direction output input
        type reset
    }
# close the bundle
}
```

6 Core RTL Configuration File

The required core RTL configuration file provides a description of the core and its interfaces. The name of the file needs to be <corename>_rtl.conf, where corename is the name of the module to be used. For example, the file defining a core named uart must be called uart_rtl.conf.

For a description of the lexical grammar, see page 59.

Syntax

The core RTL configuration file is written using standard Tcl syntax. Syntax is described using the following conventions:

[ ] optional construct
| or, alternate constructs
* zero or more repetitions
+ one or more repetitions
<> enclose names of syntactic units
() are used for grouping
{ } are part of the format and are required. An open brace must always appear on the same line as the statement
comments

The syntax for the core RTL configuration file is:

```
version <version_string>
module <core_name> {<core_stmt>+}
```

core_name is the name of the core being described and:

```
<core_stmt>:
    | icon <file_name>
    | core_id <vendor_code> <core_code> <revision_code>
```

```
  [<description>]
| interface <interface_name> bundle <bundle_name>
    [{<interface_body>*}]
| addr_region <name> {<addr_region_body>*}
| proprietary <vendor_code> <organization_name>
    {<proprietary_statements>}
```

The file must contain a single version statement followed by a single module statement. The module statement contains multiple core statements. One core_id must be included. At least one interface statement must be included. One icon statement and one or more addr_region and proprietary statements may also be included.

Components

This section describes the core RTL configuration file components.

Version Statement

The version statement identifies the version of the core RTL configuration file format. The version string consists of major and minor version numbers separated by a period. The current version of the file is 2.5.

Icon Statement

This statement specifies the icon to display on a core. The syntax is:

```
icon <file_name>
```

file_name is the name of the graphic file, without any directory names. Store the file in the design directory of the core. For example:

```
icon "myCore.ppm"
```

The supported graphic formats are GIF, PPM, and PGM. Graphics should be no larger than 80x80 pixels. Since the text used for the core is white, use a dark background for your icon, otherwise it will be difficult to read.

Core_Id Statement

The core_id statement provides identifying information to the tools about the core. This information is required. Syntax of the core_id statement is:

```
core_id <vendor_code> <core_code> <revision_code> [<description>]
```

where:

vendor_code — An OCP-IP-assigned vendor code that uniquely identifies the core developer. OCP-IP maintains a registry of assigned vendor codes. The allowed range is 0x0000 - 0xFFFF. Use 0x5555 to denote an anonymous vendor. For a list of codes check www.ocpip.org.

core_code — A developer-assigned core code that (in combination with the vendor code) uniquely identifies the core. OCP-IP provides suggested values for common cores. Refer to "Defined Core Code Values". The allowed range is 0x000 - 0xFFF.

revision_code — A developer-assigned revision code that can provide core revision information. The allowed range is 0x0 - 0xF.

description — An optional Tcl string that provides a short description of the core.

Defined Core Code Values

```
0x000 - 0x7FF: Pre-defined
    0x000 - 0x0FF: Memory
   Sum values from following choices:
      ROM:
      0x0: None
      0x1: ROM/EPROM
      0x2: Flash (writable)
      0x3: Reserved
      SRAM:
      0x0: None
      0x4: Non-pipelined SRAM
      0x8: Pipelined SRAM
      0xC: Reserved
      DRAM:
      0x00: None
      0x10: DRAM (trad., page mode, EDO, etc.)
      0x20: SDRAM (all flavors)
      0x30: RDRAM (all flavors)
      0x40: Several
      0x50: Reserved
      0x60: Reserved
      0x70: Reserved
      Built-in DMA:
      0x00: No
      0x80: Yes
   Values from 0x000 - 0x0FF are defined/reserved
   Example: Memory controller supporting only SDRAM & Flash
      would have <cc> = 0x2 + 0x20 = 0x022

    0x100 - 0x1FF: General-purpose processors
   Sum values from following choices plus offset 0x100:
      Word size:
      0x0: 8-bit
      0x1: 16-bit
      0x2: 32-bit
      0x3: 64-bit
      0x4 - 0x7: Reserved
      Embedded cache:
```

```
        0x0: No cache
        0x8: Cache (Instruction, Data, combined, or both)
        Processor Type:
        0x00: CPU
        0x10: DSP
        0x20: Hybrid
        0x30: Reserved
    Only values from 0x100 - 0x13F are defined/reserved
    Example: 32-bit CPU with embedded cache
        would have <cc> = 0x100 + 0x2 + 0x8 + 0x00 = 0x10A

    0x200 - 0x2FF: Bridges
    Sum values from following choices plus offset 0x200:
       Domain:
       0x00 - 0x7F: Computing
          0x00 - 0x3F: PC's
          0x00: ISA (inc. EISA)
          0x01 - 0x0F: Reserved
          0x10: PCI (33MHz/32b)
          0x11: PCI (66MHz/32b)
          0x12: PCI (33MHz/64b)
          0x13: PCI (66MHz/64b)
          0x14 - 0x1F: AGP, etc.
          0x40 - 0x7F: Reserved
       0x80 - 0xBF: Telecom
          0xA0 - 0xAF: ATM
          0xA0: Utopia Level 1
          0xA1: Utopia Level 2
          ...
       0xC0 - 0xFF: Datacom

    0x300 - 0x3FF: Reserved

    0x400 - 0x5FF: Other processors
    (enumerate types: MPEG audio, MPEG video, 2D Graphics,
    3D Graphics, packet, cell, QAM, Vitterbi, Huffman,
    QPSK, etc.)

    0x600 - 0x7FF: I/O
    (enumerate types: Serial UART, Parallel, keyboard, mouse,
    gameport, USB, 1394, Ethernet 10/100/1000, ATM PHY,
    NTSC, audio in/out, A/D, D/A, I2C, PCI, AGP, ISA,
    etc.)

0x800 - 0xFFF: Vendor-defined
        (explicitly left up to vendor)
```

Interface Statement

The interface statement defines and names the interfaces of a core. The interface name is required so that cores with multiple interfaces can specify to which interface a particular connection should be made. Syntax for the interface statement is:

```
interface <interface_name> bundle <bundle_name>
[{<interface_body>+}]
```

An interface statement must contain a single interface_type statement and at least one port statement. All other interface body statements are optional

The <bundle_name> must be a defined bundle such as OCP or a bundle specified in an interface configuration file as described on page 59.

In the following example, an interface named xyz is defined as an OCP bundle. The quotation marks around xyz are not required but help to distinguish the format.

```
interface "xyz" bundle ocp
```

```
<interface_body>:
      | bundle_version <version_string>
      | interface_type <type_name>
      | port <port_name> net <net_name>
      | reference_port <interface_name>.<port_name> net <net_name>
      | prefix <name>
      | param <name> <value> [{(<attribute> <value>)*}]
      | subnet <net_name> <bit_range> <subnet_name>
      | location (n|e|w|s|) <number>
      | proprietary <vendor_code> <organization_name>
        {<proprietary_statements>}

<bit_range>:
      | <bit_number>[:<bit_number>]
```

Ports on a core interface may have names that are different than the nets defined in the bundle type for the interface. In this case, each port in the interface must be mapped to the net in the bundle with which it is associated. Mapping links the module port <prefix><port_name> with the bundle <net_name>.

The default rules for mapping are that the port_name is the same as the net_name and the prefix is the name of the interface. These rules can be overridden using the Port and Prefix statements.

Bundle_version Statement

Where more than one revision exists, the bundle_version statement defines the version of the bundle to use. Syntax for the bundle_version statement is:

```
bundle_version <version_string>
```

The version_string consists of major and minor version numbers separated by a period. If no bundle_version statement is present, the default version 1.0 is used. The current version of the OCP bundle is 2.0.

Interface_type Statement

The interface_type statement defines characteristics of the bundle. Typically, the different types specify whether the core drives or receives a particular signal within the bundle. Syntax for the interface_type statement is:

```
interface_type <type_name>
```

The type_name must be a type defined in the bundle definition. If the bundle is OCP, the allowed types are: master, system_master, slave, system_slave, and monitor as described in Table 14 on page 28. To define a type, specify it in the interface configuration file (described on page 59).

Port Statement

Use the port statement to map a single port corresponding to a signal that is defined in the bundle. Syntax for the port statement is:

```
port <port_name> net <net_name>
```

The module port named <prefix><port name> implements the <net_name> function of the bundle. The legal net_name values are defined in the bundle definition. For OCP bundles, the net names are defined in "Signals and Encoding" on page 11.

Reference_port Statement

The reference_port statement re-directs a net to another bundle. Syntax for the port statement is:

```
reference_port <interface_name>.<port_name> net <net_name>
```

The interface (in which the reference_port is declared) does not have the reference port and the bundle does not have the reference net. The reference_port statement declares that the net is internally connected to the given port of the referenced interface. For example, consider the following two interfaces:

```
interface tp bundle ocp {
   reference_port ip.Clk_i net Clk
   reference_port ip.SReset_ni net MReset_n
   port Control_i net Control
   port MCmd_i net MCmd
}

interface ip bundle ocp {
   port Clk_i net Clk
   port SReset_ni net SReset_n
   port Control_i net Control
   port MCmd_o net MCmd
}
```

Figure 7 Reference Port

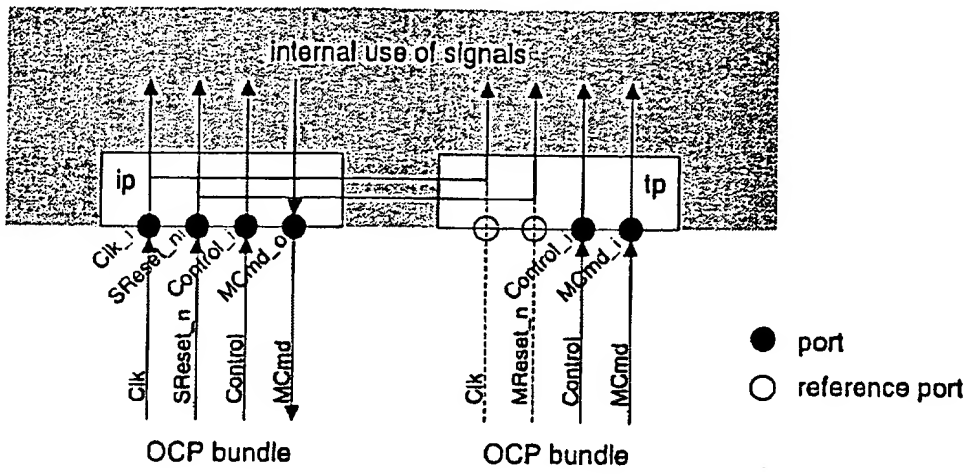


Figure 7 illustrates the operation of a reference port. In the interface tp, no ports exist for bundle signals Clk and MReset_n. Neither do the bundle signals themselves exist. Instead, they reference the corresponding ports in the ip interface and nets in the bundle connected to that interface. The internal signals in the tp interface that would have been connected to the Clk and MReset_n signals of the OCP bundle connected to the tp interface are instead connected to the referenced ports in the ip interface.

Prefix Command

The prefix command applies to all ports in an interface. It supplies a string that serves as the prefix for all core port names in the interface. Syntax for the prefix command is:

```
prefix <name>
```

For example, the statement prefix `external_` specifies that the names for all ports in the interface are of the form `external_*`.

If the prefix command is omitted, the interface name will be inserted as the default prefix. To omit the prefix from the port name, specify it as an empty string, that is `prefix ""`.

Configurable Interfaces Parameters

For configurable interfaces, parameters specify configurations. The specific parameters for OCP are described in Chapters 3 and 4 and summarized in Table 22 on page 54. The syntax for setting a parameter is:

```
param <name> <value> [{(<attribute> <value>)*}]
<value>: <number>|<name>
<attribute>: tie_off|width
```

If the parameter is used to configure a signal, the attribute list can be used to attach additional values to that signal. The supported attributes are the tie-off (if the signal is configured out of the interface) and the signal width (if the

signal is configured into the interface). Specifying the signal width using an attribute attached to the signal parameter is equivalent to using the corresponding signal width parameter but the attribute syntax is preferred.

For example, an OCP might be configured to include an interrupt signal as follows.

```
param interrupt 1
```

The following example shows the MBurstLength field tied off to a constant value of 4.

```
param burstlength 0 {tie_off 4}
```

The following code shows two equivalent ways of setting the address width to 16 bits though the second method is preferred.

```
param addr_wdth 16

param addr 1 {width 16}
```

Subnet Statement

The subnet statement assigns names to bits or contiguous bit-fields within a net. Syntax for the subnet statement is:

```
subnet <net_name> <bit_range_list> <subnet_name>
<bit_range_list>: <bit_range>[,<bit_range>]*
<bit_range>: <bit_number>[:<bit_number>]
```

The subnet_name is assigned to the bit_range within the given net_name. Bit_range can be either a single bit or a range. Subnet_name is a Tcl string.

For example bit 3 of the MReqInfo net may be assigned the name "cacheable" as follows:

```
subnet MReqInfo 3 cacheable
```

Location Statement

The location statement provides a way for the core to indicate where to place this interface when a schematic symbol for the core is drawn. The location is specified as a compass direction of north(n), south(s), east(e), west(w) and a number. The number indicates a percentage from the top or left edge of the block. Syntax for the location statement is:

```
location (n|e|w|s) <number>
```

To place an interface on the bottom (south-side) in the middle (50% from the left edge) of the block, for example, use this definition:

```
location s 50
```

Address Region Statement

The address region statement specifies address regions within the complete address space of a core. It allows you to give a symbolic name to a region, and to specify its base, size, and behavior.

```
addr_region <name> {<addr_region_body>*}
```

where:

```
<addr_region_body>: addr_base <integer> | addr_size <integer>
                  | addr_space <integer>
                  | proprietary <vendor_code> <organization_name>
                    {<proprietary_statements>}
```

- The addr_base statement specifies the base address of the region being defined and is specified as an integer.
- The addr_size statement similarly specifies the size of the region.
- The addr_space statement specifies to which OCP address space the region belongs. If the addr_space statement is omitted, the region belongs to all address spaces.

Proprietary Statement

The proprietary statement enables proprietary extensions of the core RTL configuration file syntax. Standard parsers must be able to ignore the extensions, while proprietary parsers can extract additional information about the core. Syntax for the proprietary statement is:

```
proprietary <vendor_code> <organization_name>
      {<proprietary_statements>}
```

The vendor_code uniquely identifies the vendor associated with the proprietary extensions and is described in more detail on page 66.

The organization_name specifies the name of the organization that specified the extensions. Any number of proprietary statements can be included between the braces but must follow legal Tcl syntax.

The proprietary statement can be included at multiple levels of the syntax hierarchy, allowing it to use scoping to imply context. If multiple proprietary statements are included in a single scope, the parser must process these in an additive fashion.

Sample RTL Configuration File

The format for a core RTL configuration file for a core is shown in Example 1.

Example 1 Sample flashctrl_rtl.conf File

```
# define the module
version 2.5

module flashctrl {
core_id 0xBBBB 0x001 0x1 "Flash/Rom Controller"

   # Use the Vista icon
   icon "vista.ppm"

   addr_region "FLASHCTRL0" {
   addr_base 0x0
   addr_size 0x100000
   }

   # one of the interfaces is an OCP slave using the pre-defined OCP bundle
   interface tp bundle ocp {

   # version OCP 2.0 is used
   bundle_version 2.0

   # this is a slave type ocp interface
   interface_type slave

        # this OCP is a basic interface with byteen support plus a named SFlag
        # and MReset_n
        param mreset 1
        param sreset 0
        param byteen 1
        param sflag 1 {width 1}
        param addr 1 {width 32}
        param mdata 1 {width 64}
        param sdata 1 {width 64}

        prefix tp
        # since the signal names do not exactly match the signal
        # names within the bundle, they must be explicitly linked
        port Reset_ni          net MReset_n
        port Clk_i             net Clk
        port TMCmd_i           net MCmd
        port TMAddr_i          net MAddr
        port TMByteEn_i        net MByteEn
        port TMData_i          net MData
        port TCCmdAccept_o     net SCmdAccept
        port TCResp_o          net SResp
```

```
        port TCData_o          net SData
        port TCError_o         net SFlag

        # name SFlag[0] access_error
        subnet SFlag 0 access_error

        # stick this interface in the middle of the top of the module
        location n 50

    ) # close interface tp defininition


    # The other interface is to the flash device defined in an interface file
    # Define the interface for the Flash control
    interface emem bundle flash {

       # the type indicates direction and drive of the control signals
        interface_type controller

       # since this module has direction indication on some of the signals
       # ('_o','_b') and is missing assertion level indicators '_n' on
       # some of the signals, the names must again be directly linked to
       # the signal names within the bundle
        port Addr_o    net addr
        port Data_b    net data
        port OE        net oe_n
        port WE        net we_n
        port RP        net rp_n
        port WP        net wp_n

        # all of the signals on this port have the prefix 'emem_'
        prefix "emem_"

        # stick this interface in the middle of the bottom of the module
        location s 50

    } # close interface emem defininition

} # close module definition
```

The flash bundle is defined in the following interface configuration file. See "Interface Configuration File" on page 59 for the syntax definition of the interface configuration file.

```
bundle flash {
    #types of flash interfaces
    #controller: flash controller; flash: flash device itself.
    interface_types controller flash
    net addr {
        #Address to the Flash device
```

```
            direction output input
            width 19
        }
        net data {
            #Read or Write Data
            direction inout inout
            width 16
        }
        net oe_n {
            # Output Enable, active low.
            direction output input
        }
        net we_n {
            # Write Enable, active low.
            direction output input
        }
        net rp_n {
            # Reset, active low.
            direction output input
        }
        net wp_n {
            # Write protect bit,  Active low.
            direction output input
        }
}
```

7 Core Timing

To connect two entities together, allowing communication over an OCP interface, the protocols, signals, and pin-level timing must be compatible. This chapter describes how to define interface timing for a core. This process can be applied to OCP and non-OCP interfaces.

Use the core synthesis configuration file to set timing constraints for ports in the core. The file consists of any of the constraint sections: port, max delay, and false path. If the core has additional non-OCP clocks, the file should contain their definitions.

When implementing IP cores in a technology independent manner it is difficult to specify only one timing number for the interface signals, since timing is dependent on technology, library and design tools. The methodology specified in this chapter allows the timing of interface signals to be specified in a technology independent way.

To make your core description technology independent use the technology variables defined in the *Core Preparation Guide*. The technology variables range from describing the default setup and clock-to-output times for a port to defining a high drive cell in the library.

Timing Parameters

There is a set of minimum timing parameters that must be specified for a core interface. Additional optional parameters supply more information to help the system designer integrate the core. Hold-time parameters allow hold time checking. Physical-design parameters provide details on the assumptions used for deriving pin-level timing.

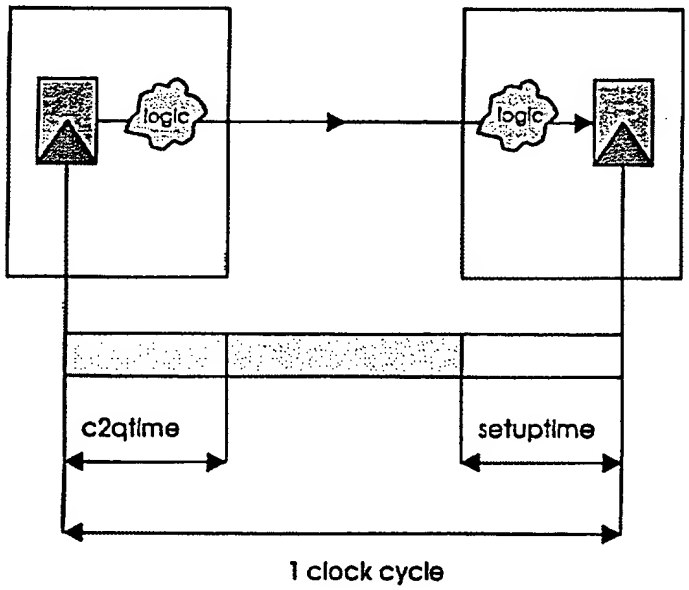
Minimum Parameters

At a minimum, the timing of an OCP interface is specified in terms of two parameters:

- `setuptime` is the latest time an input signal is allowed to change before the rising edge of the clock.
- `c2qtime` is the latest time an output signal is guaranteed to become stable after the rising edge of the clock.

Figure 8 shows the definition of `setuptime` and `c2qtime`. See "Port Constraint Keywords" on page 85 for a description of these parameters.

Figure 8 OCP Timing Parameters



Hold-time Parameters

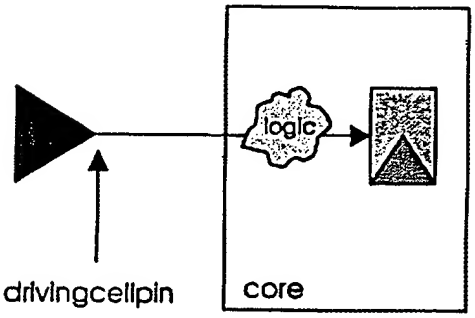
Hold-time parameters are needed to allow the system integrator to check hold time requirements. On the output side, `c2qtimemin` specifies the minimum time for a signal to propagate from a flip-flop to the given output pin. On the input side, `holdtime` specifies the minimum time for a signal to propagate from the input pin to a flip-flop.

Technology Variables

To give meaning to the timing values, timing requirements on input and output pins must be accompanied by information on the assumed environment for which these numbers are determined. This information also adds detail on the expected connection of the pin.

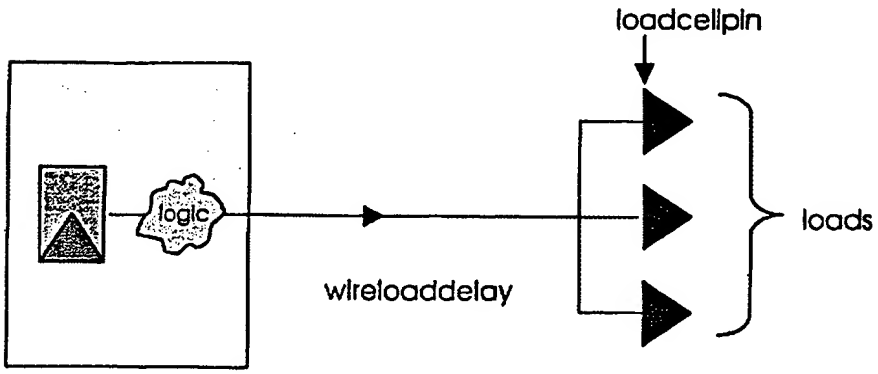
For an input signal, the parameter `drivingcellpin` indicates the cell library name for a cell representative of the strength of the driver that needs to be used to drive the signal. This is shown in Figure 9.

Figure 9 Driver Strength



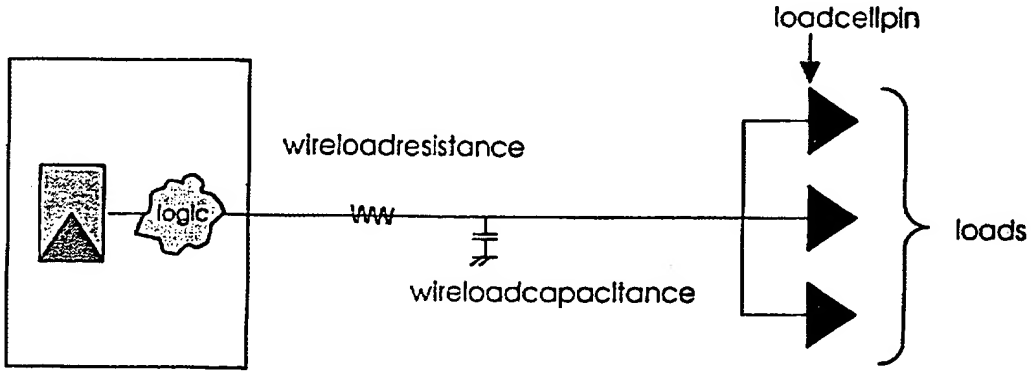
For an output signal, the variable `loadcellpin` indicates the input load of the gate that the signal is expected to drive. The variable `loads` indicates how many loadcellpins the signal is expected to drive. Additionally, information on the capacitive load of the wire must be included. There are two options. Either the variable `wireloaddelay` can be specified, as shown in Figure 10. Or, the combination `wireloadcapacitance/wireloadresistance` must be specified, as shown in Figure 11.

Figure 10 Variable Loads - wireloaddelay



For instructions on calculating a delay, refer to the *Synopsys Design Compiler Reference*.

Figure 11 Variable Loads - wireloadresistance/wireloadcapacitance



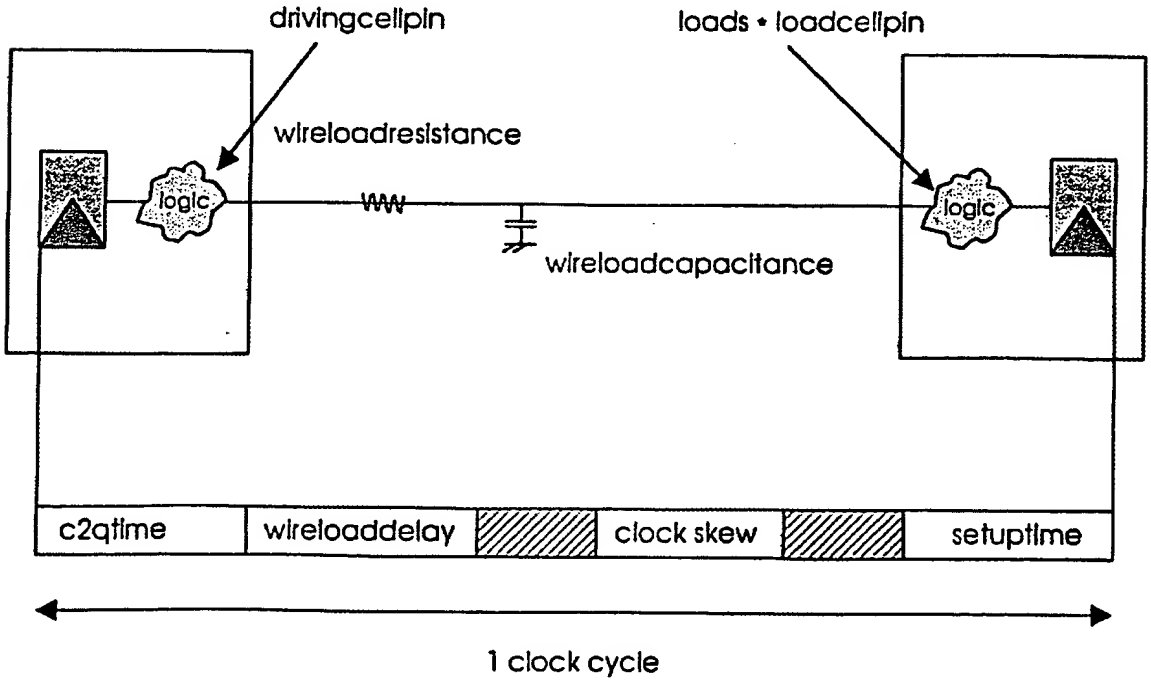
Connecting Two OCP Cores

Figure 12 shows the timing model for interconnecting two OCP compliant cores.

The sum of `setuptime`, `c2qtime` and wire delay must be less than the clock period or cycle time minus the clock-skew. Similarly, the minimum clock-cycle for two cores to interoperate is determined by the maximum of the sum of `c2qtime`, `setuptime`, wire delay and clock-skew over all interface signals.

The wireload delay is defined by either the variable `wireloaddelay` or the set `wirelcadcapacitance/wireloadresistance`.

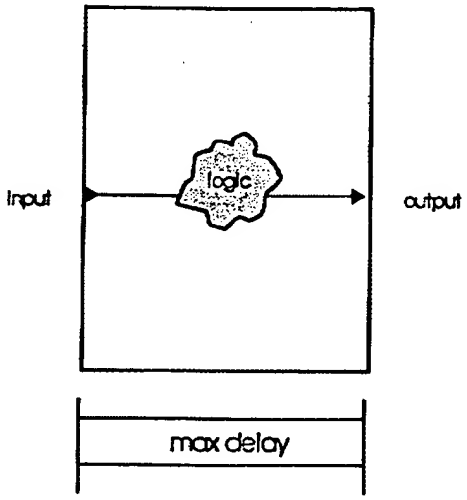
Figure 12 Connecting Two OCP Compliant Cores



Max Delay

In addition to the `setup` and `c2qtime` paths for a core, there may also be combinational paths between input and output ports. Use `maxdelay` to specify the timing for these paths.

Figure 13 Max Delay Timing



False Paths

It is possible to identify a path between two ports as being logically impossible. Such paths can be specified using the `falsepath` constraint syntax.

For instructions on specifying the core's timing parameters, see "False Path Constraints" on page 90.

Core Synthesis Configuration File

The core synthesis configuration file contains the following sections:

Version
: Specifies the current version of the synthesis configuration file format. The current version is 1.3.

Clock
: Describes clocks brought into the core.

Area
: Defines the area in gates of the core.

Port
: Defines the timing of IP block ports.

Max Delay
: Specifies the delay between two ports on a combinational path.

False Path
: Specifies that a path between input and output ports is logically impossible.

Syntax Conventions

Observe the following syntax conventions:

- Enclose all expr statements within braces { }, to differentiate between expressions that are to be evaluated while the file is being parsed (without braces) and those that are to be evaluated during synthesis constraint file generation (with braces).
- Although not required by Tcl, enclose strings within quotation marks "", to show that they are different than keywords.
- Specify keywords using lower case.

Parameter values are specified using Tcl syntax. Expressions can use any of the technology or environment variables, and any of the following variables:

clockperiod
: This variable should only be used in calculations of timing values for ports. When evaluating expressions that use `$clockperiod`, the program will determine which clock the port is relative to, determine its period (in nanoseconds), and apply that value to the equation. For example:

```
port "in" {
   setuptime {[expr $clockperiod * .5]}
}
```

rootclockperiod

This variable is set to the period of the main system clock, usually referred to as the root clock. It is typically used when a value needs to be a multiple of the root clock, such as for non-OCP clocks. For example:

```
clock "myClock" {
   period {[expr $rootclockperiod * 4]}
}
```

The design_syn.conf file can also use conditional settings of the parameters in the design as outlined by the following arrays. These variables are only used at the time the file is read into the tools.

param

This array is indexed by the configuration parameters that can be found on a particular instance. Only use `param` for core_syn.conf files since it is only applicable to the instance being processed. For example:

```
if { $param("dma_fd") == 1 } {
   port "T12_ipReset_no" {
        c2qtime {[expr $clockperiod * 0.7]}
   }
}
```

chipparam

This array is indexed by the configuration parameters that are defined at the chip or design level. These variables can be used in both the design_syn.conf and core_syn.conf files as they are more global in nature than those specified by `param`. For example:

```
if { $chipparam("full") == 1 } {
   instance "bigcore" {
      port "in" {
        setuptime {[expr $clockperiod * 0.7]}
      }
   }
}
```

interfaceparam

This array is indexed by the interface name and the configuration parameters that are on an interface. It should only be used for core_syn.conf files since it is only applicable to the interfaces on the instance being processed. In the following example the interface name is ip.

```
if { $interfaceparam("ip_respaccept") == 1 } {
    port "ipMRespAccept_o" {
        c2qtime {[expr $clockperiod * 21/25]}
    }
}
```

Version Section

The version of the core synthesis configuration file is required. Specify the version with the version command, for example: `version 1.3`

Clock Section

If you have non-OCP clocks for an IP block or want to specify the `worstcasedelay` of any clock (including OCP clocks) used in the core, specify the names of the clocks in the core synthesis configuration file. Use the following syntax to specify the name of the clock and its `worstcasedelay`:

```
clock <clockName> {
   worstcasedelay <delay Value>
}
```

`clockName` refers to the name of the port that brings the clock into the core for the core synthesis configuration file. For example:

```
clock "myClock"
```

worstcasedelay
: The worst case delay value is the longest path through the core or instance for a particular clock. The value is used to check that the core can meet the timing requirements of the current design. To help make this value more portable, you may want to use the technology variable `gatedelay`. For example:

```
clock "myClock" {
   worstcasedelay {[10.5 * $gatedelay]}
}

clock "otherClock" {
   worstcasedelay 5
}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

Area Section

The area is the size in gates of the core or instance. By specifying the size in gates the area can be calculated based on the size of a typical two input nand gate in a particular synthesis library. For example:

```
area {[expr 20500 / $gatesize]}
area 5000
```

Constant values are specified in two input nand gate equivalents. For consistency, use expression that can be interpreted in gates.

Port Constraints Section

Use the port constraints section to specify the timing parameters. Input port information that can be specified includes the setup time, related clock (non-OCP ports), and driving cell. For output ports, the clock to output times, related clock (non-OCP ports), and the loading information must be supplied.

Port Constraint Keywords

The keywords that can be used to specify information about port constraints are:

c2qtime
: The c2q (clock to q or clock to output) time is the longest path using worst case timing from a starting point in the core (register or input port) to the output port. This includes the `c2qtime` of the register. To maintain portability, most cores specify this as a percentage of the fastest clock period used while synthesizing the core. For example:

```
c2qtime {[expr $timescale * 3500]}
c2qtime {[expr $clockperiod * 0.25]}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

c2qtimemin
: The c2q (clock to q or clock to output) time min is the shortest path using best case timing from a starting point in the core (register or input port) to the output port. This includes the `c2qtime` of the register. Most cores use the default from the technology section, `defaultc2qtimemin`. For example:

```
c2qtimemin {[expr $timescale * 100]}
c2qtimemin {$defaultc2qtimemin}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

clockname
: This is an optional field for all OCP ports and is a string specifying the associated clock portname. For input ports, input delays use this clock as the reference clock. For output ports, output delays use this clock as the reference clock. For example:

```
clockname "myClock"
```

drivingcellpin
: This variable describes which cell in the synthesis library is expected to be driving the input. To maintain portability set this variable to use one of the technology values of `high/medium/lowdrivegatepin`.

Values are a string that specifies the logical name of the synthesis library, the cell from the library, and the pin that will be driving an input for the core. The pin is optional. For example:

```
drivingcellpin {$mediumdrivegatepin}
drivingcellpin "pt25u/nand2/O"
```

holdtime

The hold time is the shortest path using best case timing from an input port to any endpoint in the core. Most cores use the default from the technology section, `defaultholdtime`. For example:

```
holdtime {[expr $timescale * 100]}
holdtime {$defaultholdtime}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

loadcellpin

The name of the load library/cell/pin that this output port is expected to drive. The value is specified to the synthesis tool as the gate to use (along with the number of loads) in its load calculations for output ports of a module. For portability use the default.

Values are a string that specifies the logical name of the synthesis library, the cell from the library, and the pin that the load calculation is derived from. The pin is optional. For example:

```
loadcellpin "pt25u/nand2/I1"
loadcellpin {$defaultloadcellpin}
```

loads

The number of loadcellpins that this output port is expected to drive. The value is communicated to the synthesis tool as the number of loads to use in load calculations for output ports of a module. The typical setting for this is the technology value of `defaultloads`. Values are an expression that evaluates to an integer. For example:

```
loads 5
loads {$defaultloads}
```

maxfanout

This keyword limits the fanout of an input port to a specified number of fanouts. To maintain portability set this variable in terms of the technology variable `defaultfanoutload`. Constant values are specified in library units. For example:

```
maxfanout {[expr $defaultfanoutload * 1]}
```

setuptime

The longest path using worst case timing from an input port to any endpoint in the core. To maintain portability, most cores specify this as a percentage of the fastest clock period used during synthesis of the core. For example:

```
setuptime {[expr $timescale * 2500]}
setuptime {[expr $clockperiod * 0.25]}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

wireloaddelay
Replaces capacitance/resistance as a way to specify expected delays caused by the interconnect. To maintain portability set this variable to use a technology value of `long/medium/shortnetdelay`.

The resulting values get added to the worst case clock-to-output times of the ports to anticipate net delays of connections to these ports. To improve the accuracy of the delay calculation cores should use the resistance and capacitance settings.

You cannot specify both `wireloaddelay` and `wireloadresistance/capacitance` for the same port. For example:

```
wireloaddelay {[expr $clockperiod * .25]}
wireloaddelay {$mediumnetdelay}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

wireloadresistance
wireloadcapacitance
Specify expected loading and resistance caused by the interconnect. If available, specify both resistance and capacitance. To maintain portability set this variable to use one of the technology values of `long/medium/shortnetrcresistance/capacitance`.

If these constraints are specified they show up as additional loads and resistances on output ports of a module. You cannot use both wireloaddelay and `wireloadresistance/capacitance` for the same port.

Specify constant values as expressions that result in kOhms for resistance and picofarads (pf) for capacitance. For example:

```
wireloadresistance {[expr $resistancescale * .09]}
wireloadcapacitance {[expr $capacitancescale * .12]}
wireloadresistance {$mediumnetrcresistance}
wireloadcapacitance {$mediumnetrccapacitance}
```

Input Port Syntax

For input and inout ports (*inout* ports have both an input and an output definition) use the following syntax:

```
port <portName> {
   clockname <clockName>
   drivingcellpin <drivingCellName>
   setuptime <Value>
   holdtime <Value>
   maxfanout <Value>
}
```

Examples

In the following example, the clock is not specified since this is an OCP port and is known to be controlled by the OCP clock. If a clock were specified as something other than the OCP clock, an error would result.

```
port "MCmd_i" {
   drivingcellpin {$mediumdrivegatepin}
   setuptime {[expr $clockperiod * 0.2]}
}
```

In the following example, the setup time is required to be 2ns. Time constants are assumed to be in nanoseconds. Use the `timescale` variable to convert library units to nanoseconds.

```
port "MAddr_i" {
   drivingcellpin {$mediumdrivegatepin}
   setuptime 2
}
```

The following example shows how to associate a non OCP clock to a port. The example uses `maxfanout` to limit the fanout of myInPort to 1. If the logic for myInPort required it to fanout to more than one connection, the synthesis tool would add a buffer to satisfy the `maxfanout` requirement.

```
port "myInPort" {
   clockname "myClock"
   drivingcellpin {$mediumdrivegatepin}
   setuptime 2
   maxfanout {[expr $defaultfanoutload * 1]}
}
```

Output Port Syntax

For output and inout ports (inout ports have both an input and an output definition) use the following syntax:

```
port <portName> {
   clockname <clockName>
   loadcellpin <loadCellPinName>
   loads <Value>
   wireloadresistance <Value>
   wireloadcapacitance <Value>
   wireloaddelay <Value>
   c2qtime <Value>
   c2qtimemin <Value>
}
```

You cannot specify both `wireloaddelay` and `wireloadresistance/capacitance` for the same port.

Examples

In the following example, the clock is not specified since this is an OCP port and is known to be controlled by the OCP clock.

```
port "SCmdaccept_o"
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloadresistance {$mediumnetrcresistance}
   wireloadcapacitance {$mediumnetrccapacitance}
   c2qtime {[expr $clockperiod * 0.2]}
}
```

In the following example, the clock to output time is required to be 2 ns. Time constants are assumed to be in nanoseconds. Use the `timescale` variable to convert library units to nanoseconds.

```
port "SResp_o"
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloadresistance {$mediumnetrcresistance}
   wireloadcapacitance {$mediumnetrccapacitance}
   c2qtime 2
}
```

The following example shows how to associate a clock to an output port.

```
port "myOutPort"
   clockname "myClock"
   loadcellpin {$defaultloadcellpin}
   loads 10
   wireloaddelay {$longnetdelay}
   c2qtime {[expr $clockperiod * .2]}
}
```

InOut Port Example

```
port "Signal_io"
   drivingcellpin {$mediumdrivegatepin}
   setuptime {[expr $clockperiod * 0.2]}
}
port "Signal_io"
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloadresistance {$mediumnetrcresistance}
   wireloadresistance {$mediumnetrccapacitance}
   c2qtime {[expr $clockperiod * 0.2]}
}
```

Max Delay Constraints

Using the max delay constraints you can specify the delay between two ports on a combinational path. This is useful when synthesizing two communicating OCP interfaces. The syntax for `maxdelay` is:

```
maxdelay {
   delay <delayValue> fromport <portName> toport <portName>
   .
   .
   .
}
```

where: `<delayValue>` can be a constant or a Tcl expression.

In the following example, a `maxdelay` of 3 ns is specified for the combinational path between myInPort1 and myOutPort1. A `maxdelay` of 50% of the clockperiod is specified for the path between myInPort2 and myOutPort2. The braces around the expression delay evaluation until the expression is used by the mapping program.

```
maxdelay {
   delay 3 fromport "myInPort1" toport "myOutPort1
   delay {[expr $clockperiod *.5]} fromport "myInPort2" toport "myOutPort2"
}
```

False Path Constraints

Using the false path constraints you can specify that a path between certain input and output ports is logically impossible.

The syntax for `falsepath` is:

```
falsepath{
   fromport <portName> toport <portName>
.
.
.
}
```

In the following example, a `falsepath` is set up between myInPort1 and myOutPort1 as well as myInPort2 and myOutPort2. This tells the synthesis tool that the path is not logically possible and so it will not try to optimize this path to meet timing.

```
falsepath {
   fromport "myInPort1" toport "myOutPort1"
   fromport "myInPort2" toport "myOutPort2"
}
```

Sample Core Synthesis Configuration File

The following example shows a complete core synthesis configuration file.

```
version 1.3
port "Reset_ni" {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port "MCmd_i" {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .9]}
}
port "MAddr_i" {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port "MWidth_i" {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port "MData_i" {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port "SCmdAccept_o" {
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloaddelay {$mediumnetdelay}
   c2qtime {[expr $clockperiod * .9]}
}
port "SResp_o" {
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloaddelay {$mediumnetdelay}
   c2qtime {[expr $clockperiod * .8]}
}
port "SData_o" {
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloaddelay {$mediumnetdelay}
   c2qtime {[expr $clockperiod * .8]}
}
maxdelay {
   delay 2 fromport "MData_i" toport
   "SResp_o"
}
falsepath {
   fromport "MData_i" toport "SData_o"
}
```

Part II *Guidelines*

8 Timing Diagrams

The timing diagrams within this chapter look at signals at strategic points and are not intended to provide full explanations but rather, highlight specific areas of interest. The diagrams are provided solely as examples. For related information about phases, see "Signal Timing and Protocol Phases" on page 33.

Most fields are unspecified whenever their corresponding phase is not asserted. This is indicated by the striped pattern in the waveforms. For example, when MCmd is IDLE the request phase is not asserted, so the values of MAddr, MData, and SCmdAccept are unspecified.

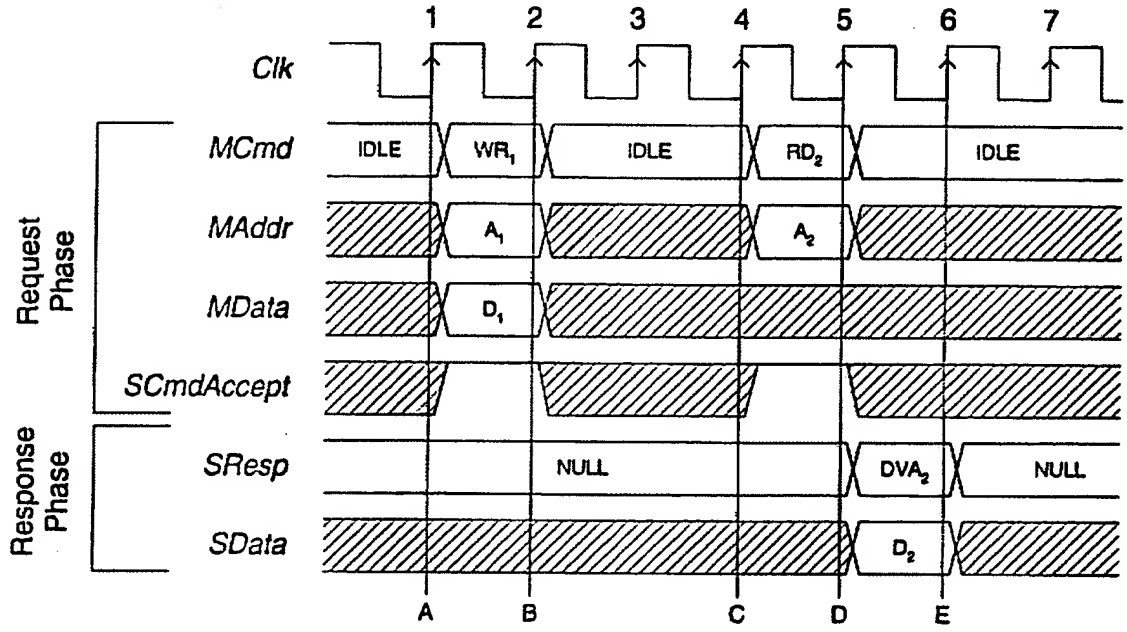
Subscripts on labels in the timing diagrams denote transfer numbers that can be helpful in tracking a transfer across protocol phases.

For a description of timing diagram mnemonics, see Tables 2 and 3 on page 14.

Simple Write and Read Transfer

Figure 14 illustrates a simple write and a read transfer on a basic OCP interface. This diagram shows a write with no response enabled on the write, which is typical behavior for a synchronous SRAM or a register bank.

Figure 14 Simple Write and Read Transfer



Sequence

A. The master starts a request phase on clock 1 by switching the MCmd field from IDLE to WR. At the same time, it presents a valid address (A1) on MAddr and valid data (D1) on MData. The slave asserts SCmdAccept in the same cycle, making this a 0-latency transfer.

B. The slave captures the values from MAddr and MData and uses them internally to perform the write. Since SCmdAccept is asserted, the request phase ends.

C. The master starts a read request by driving RD on MCmd. At the same time, it presents a valid address on MAddr. The slave asserts SCmdAccept in the same cycle for a request accept latency of 0.

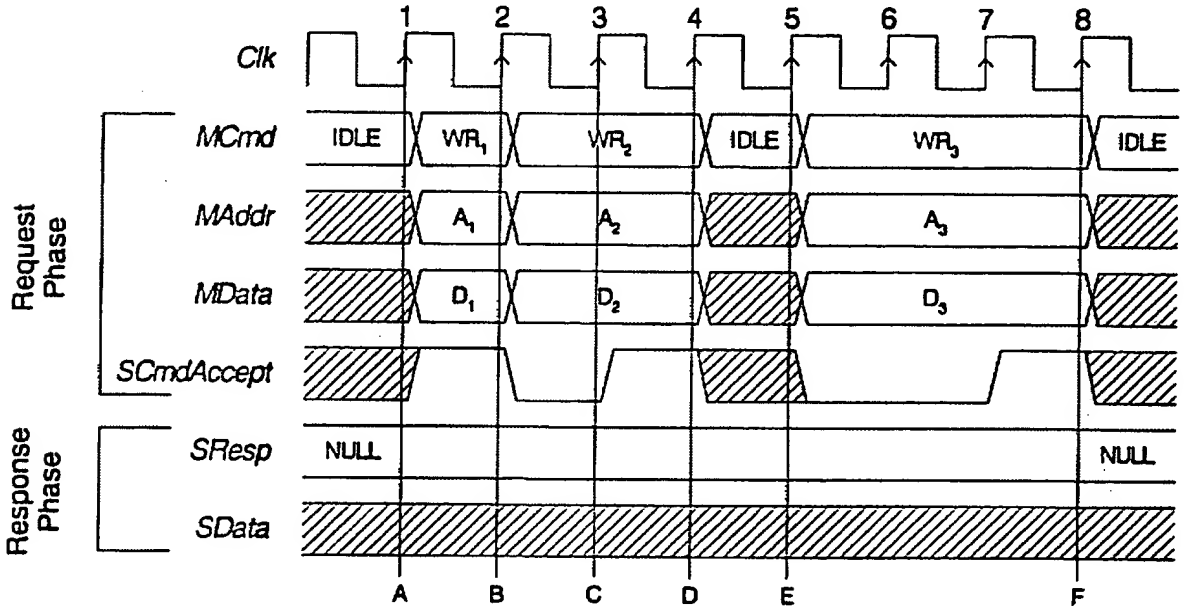
D. The slave captures the value from MAddr and uses it internally to determine what data to present. The slave starts the response phase by switching SResp from NULL to DVA. The slave also drives the selected data on SData. Since SCmdAccept is asserted, the request phase ends.

E. The master recognizes that SResp indicates data valid and captures the read data from SData, completing the response phase. This transfer has a request-to-response latency of 1.

Request Handshake

Figure 15 illustrates the basic flow-control mechanism for the request phase using SCmdAccept. There are three writes with no responses enabled, each with a different request accept latency.

Figure 15 Request Handshake



Sequence

A. The master starts a write request by driving WR on MCmd and valid address and data on MAddr and MData, respectively. The slave asserts SCmdAccept in the same cycle, for a request accept latency of 0.

B. The master starts a new transfer in the next cycle. The slave captures the write address and data. It deasserts SCmdAccept, indicating that it is not yet ready for a new request.

C. Recognizing that SCmdAccept is not asserted, the master holds all request phase signals (MCmd, MAddr, and MData). The slave asserts SCmdAccept in the next cycle, for a request accept latency of 1.

D. The slave captures the write address and data.

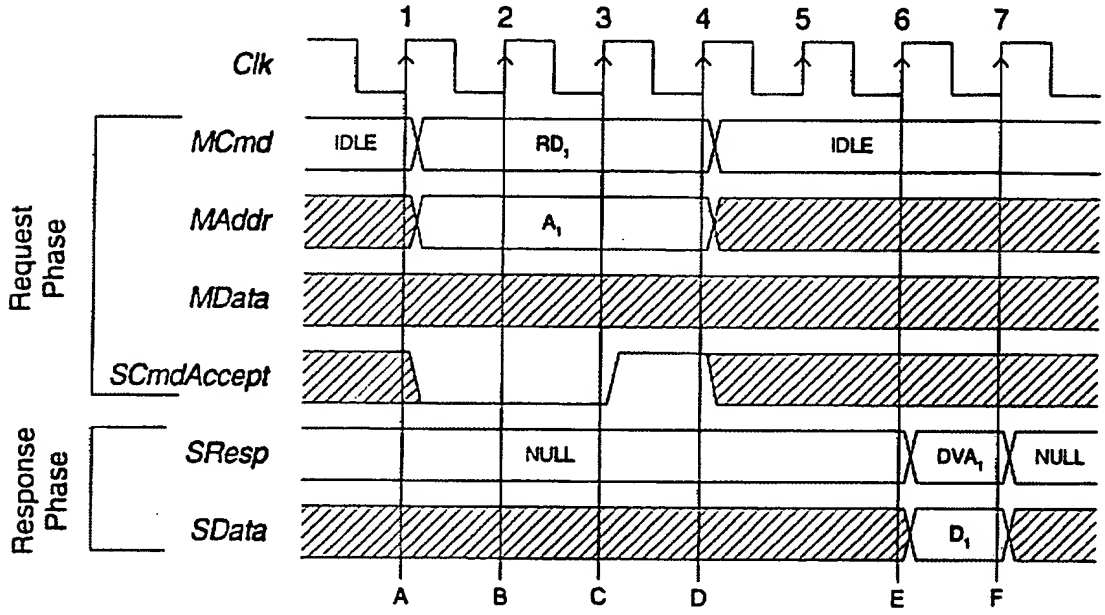
E. After 1 idle cycle, the master starts a new write request. The slave deasserts SCmdAccept.

F. Since SCmdAccept is asserted, the request phase ends. SCmdAccept was low for 2 cycles, so the request accept latency for this transfer is 2. The slave captures the write address and data.

Request Handshake and Separate Response

Figure 16 illustrates a single read transfer in which a slave introduces delays in the request and response phases. The request accept latency 2, corresponds to the number of clock cycles that SCmdAccept was deasserted. The request to response latency 3, corresponds to the number of clock cycles from the end of the request phase (D) to the end of the response phase (F).

Figure 16 Request Handshake and Separate Response



Sequence

A. The master starts a request phase by issuing the RD command on the MCmd field. At the same time, it presents a valid address on MAddr. The slave is not ready to accept the command yet, so it deasserts SCmdAccept.

B. The master sees that SCmdAccept is not asserted, so it keeps all request phase signals steady. The slave may be using this information for a long decode operation, and it expects the master to hold everything steady until it asserts SCmdAccept.

C. The slave asserts SCmdAccept. The master continues to hold the request phase signals.

D. Since SCmdAccept is asserted, the request phase ends. The slave captures the address, and although the request phase is complete, it is not ready to provide the response, so it continues to drive NULL on the SResp field. For example, the slave may be waiting for data to come back from an off-chip memory device.

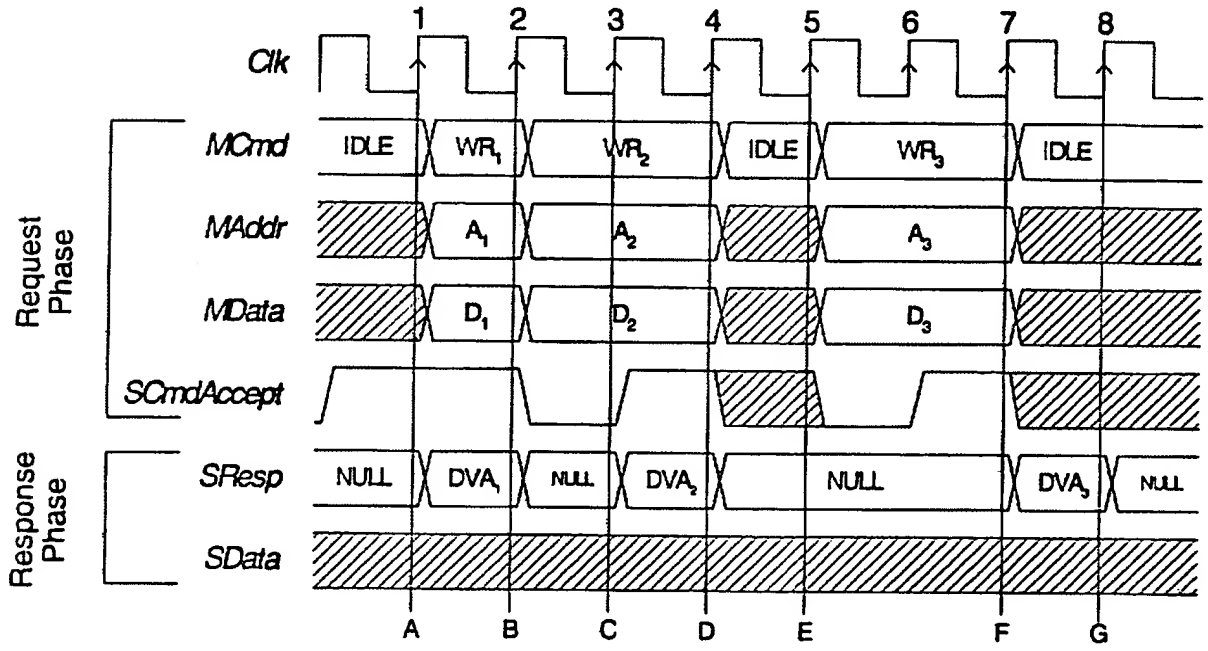
E. The slave is ready to present the response, so it issues DVA on the SResp field, and drives the read data on SData.

F. The master sees the DVA response, captures the read data, and the response phase ends.

Write with Response

Figure 17 is the same example as the waveform on page 97 but with response on write enabled. The response is typically provided to the master in the same cycle as SCmdAccept, but could be delayed (if required to perform an error check for instance). On the first write transaction, the slave uses a default accept scheme, resulting in a 0-wait state write transaction. Using fully-synchronous handshake, this condition is only possible when the slave's ability to accept a command depends solely on its internal state: any command issued by the master can be accepted. Same-cycle SCmdAccept could also be achieved using combinational logic.

Figure 17 Write with Response



Sequence

A. The master starts a write request by driving WR on MCmd and valid address and data on MAddr and MData, respectively. The slave having already asserted SCmdAccept for a request accept latency of 0, drives DVA on SResp to indicate a successful transaction.

B. The master starts a new transfer in the next cycle. The slave captures the write address and data and deasserts SCmdAccept, indicating that it is not ready for a new request.

C. With SCmdAccept not asserted, the master holds all request phase signals (MCmd, MAddr, and MData). The slave asserts SCmdAccept in the next cycle, for a request accept latency of 1 and drives DVA on SResp to indicate a successful transaction.

D. The slave captures the write address and data.

E. After 1 idle cycle, the master starts a new write request. The slave deasserts SCmdAccept.

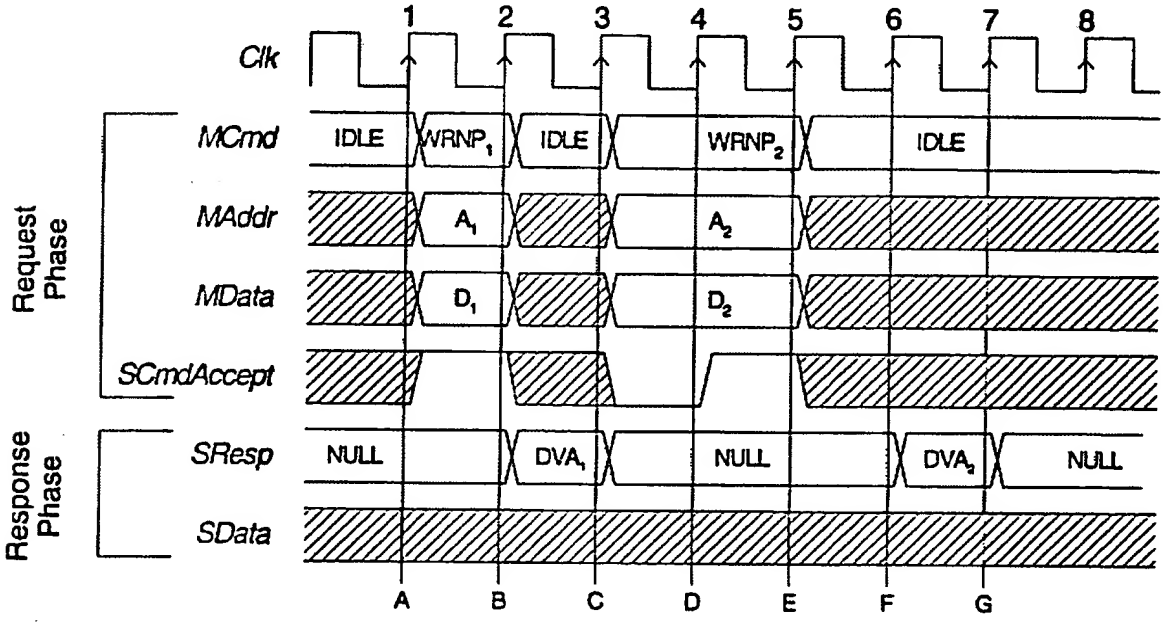
F. Since SCmdAccept is asserted, the request phase ends. SCmdAccept was low for 2 cycles, so the request accept latency for this transfer is 2. The slave captures the write address and data. The slave drives DVA on SResp to indicate a successful transaction.

G. The master samples the response.

Non-Posted Write

Figure 18 repeats the previous example for a non-posted write transaction. In this case the response must be returned to the master once the write operation occurs. There is no difference in the command acceptance, but the response may be significantly delayed. If this scheme is used for all posting-sensitive transactions, the result is decreased data throughput but higher system reliability.

Figure 18 Request Handshake



Sequence

A. The master starts a non-posted write request by driving WRNP on MCmd and valid address and data on MAddr and MData, respectively. The slave asserts SCmdAccept combinatorially, for a request accept latency of 0.

B. The slave drives DVA on SResp to indicate a successful first transaction.

C. The master starts a new transfer. The slave deasserts the SCmdAccept, indicating it is not yet ready to accept a new request. The master samples DVA on SResp and the first response phase ends.

D. The slave asserts SCmdAccept for a request accept latency of 1.

E. The slave captures the write address and data.

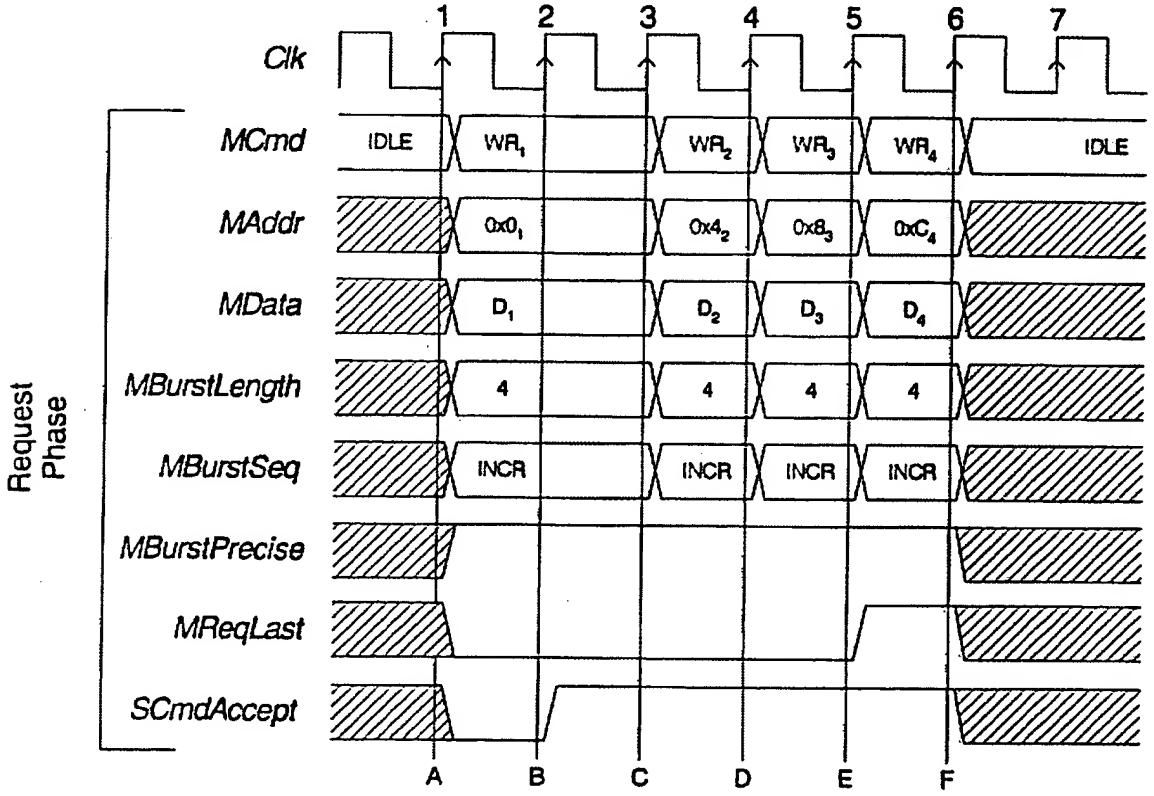
F. The slave drives DVA on SResp to indicate a successful second transaction.

G. The master samples DVA on SResp and the second response phase ends.

Burst Write

Figure 19 illustrates a burst of four 32-bit words, incrementing precise burst write, with optional burst framing information (MReqLast). As the burst is precise (with no response on write), the MBurstLength signal is constant during the whole burst. MReqLast flags the last request of the burst, and SRespLast flags the last response of the burst. The slave may either count requests or monitor MReqLast for the end of burst.

Figure 19 Burst Write



Sequence

A. The master starts the burst write by driving WR on MCmd, the first address of the burst on MAddr, valid data on MData, a burst length of four on MBurstLength, the burst code INCR on MBurstSeq, and asserts MBurstPrecise. MReqLast must be deasserted until the last request in the burst. The burst signals indicate that this is an incrementing burst of precisely four transfers. The slave is not ready for anything, so it deasserts SCmdAccept.

B. The slave asserts SCmdAccept for a request accept latency of 1.

C. The master issues the next write in the burst. MAddr is set to the next word-aligned address. For 32-bit words, the address is incremented by 4. The slave captures the data and address of the first request.

D. The master issues the next write in the burst, incrementing MAddr. The slave captures the data and address of the second request.

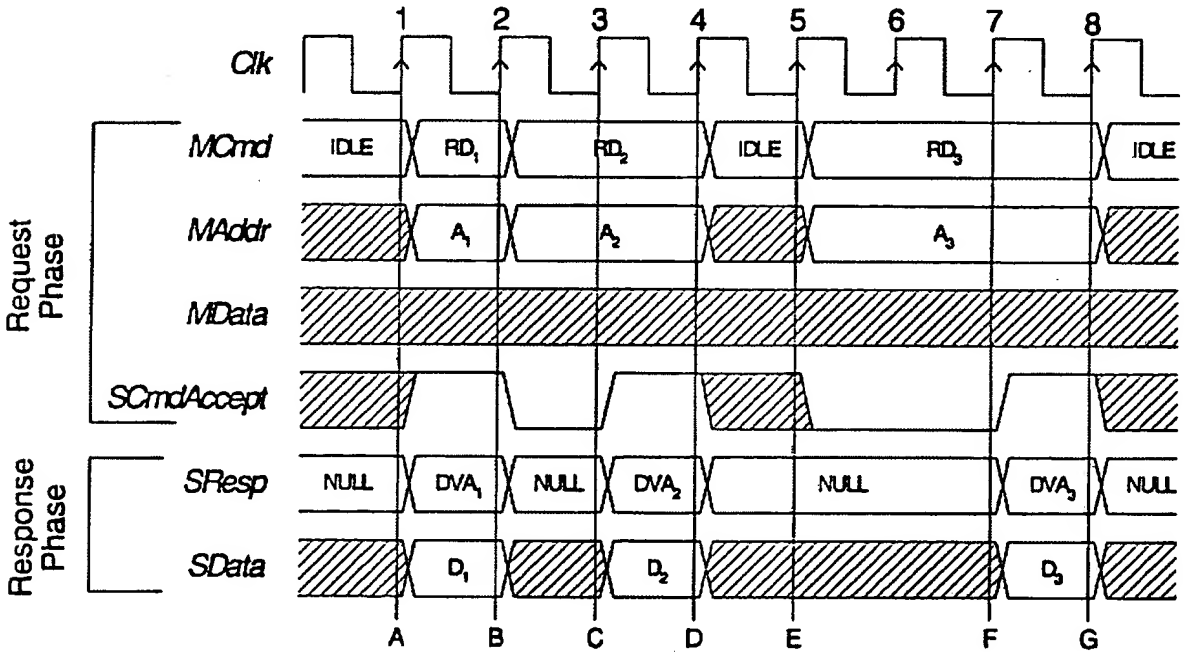
E. The master issues the final write in the burst, incrementing MAddr, and asserting MBurstLast. The slave captures the data and address of the third request.

F. The slave captures the data and address of the last request.

Non-Pipelined Read

Figure 20 shows three read transfers to a slave that cannot pipeline responses after requests. This is the typical behavior of legacy computer bus protocols with a single WAIT or ACK signal. In each transfer, SCmdAccept is asserted in the same cycle that SResp is DVA. Therefore, the request-to-response latency is always 0, but the request accept latency varies from 0 to 2.

Figure 20 Non-Pipelined Read



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The slave asserts SCmdAccept, for a request accept latency of 0. When the slave sees the read command, it responds with DVA on SResp and valid data on SData. (This requires a combinational path in the slave from MCmd, and possibly other request phase fields, to SResp, and possibly other response phase fields.)

B. The master launches another read request. It also sees that SResp is DVA and captures the read data from SData. The slave is not ready to respond to the new request, so it deasserts SCmdAccept.

C. The master sees that SCmdAccept is low and extends the request phase. The slave is now ready to respond in the next cycle, so it simultaneously asserts SCmdAccept and drives DVA on SResp and the selected data on SData. The request accept latency is 1.

D. Since SCmdAccept is asserted, the request phase ends. The master sees that SResp is now DVA and captures the data.

E. The master launches a third read request. The slave deasserts SCmdAccept.

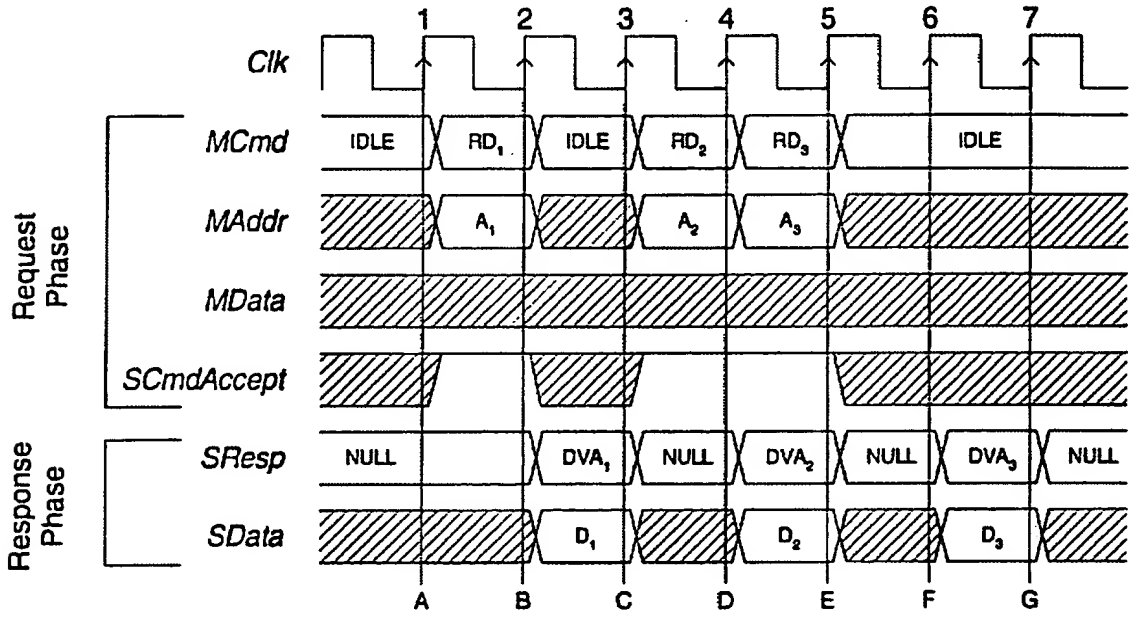
F. The slave asserts SCmdAccept after 2 cycles, so the request accept latency is 2. It also drives DVA on SResp and the read data on SData.

G. The master sees that SCmdAccept is asserted, ending the request phase. It also sees that SResp is now DVA and captures the data.

Pipelined Request and Response

Figure 21 shows three read transfers using pipelined request and response semantics. In each case, the request is accepted immediately, while the response is returned in the same or a later cycle.

Figure 21 Pipelined Request and Response



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The slave asserts SCmdAccept, for a request accept latency of 0.

B. Since SCmdAccept is asserted, the request phase ends. The slave responds to the first request with DVA on SResp and valid data on SData.

C. The master launches a read request and the slave asserts SCmdAccept. The master sees that SResp is DVA and captures the read data from SData. The slave drives NULL on SResp, completing the first response phase.

D. The master sees that SCmdAccept is asserted, so it can launch a third read even though the response to the previous read has not been received. The slave captures the address of the second read and begins driving DVA on SResp and the read data on SData.

E. Since SCmdAccept is asserted, the third request ends. The master sees that the slave has produced a valid response to the second read and captures the data from SData. The request-to-response latency for this transfer is 1.

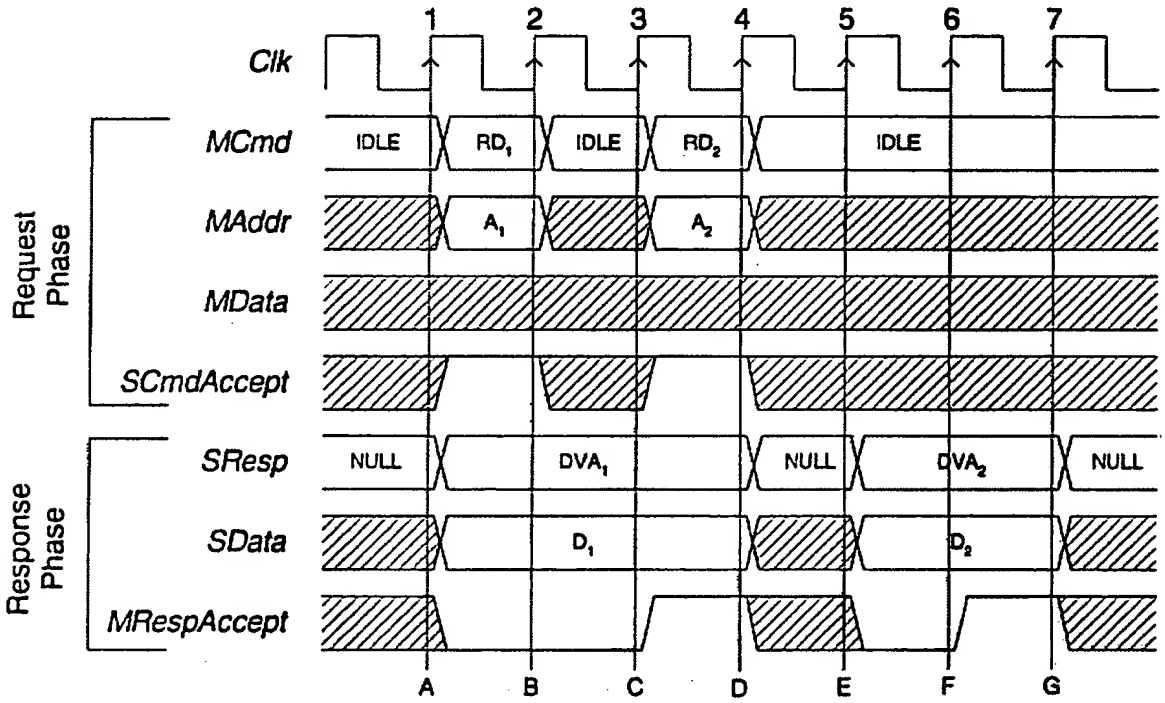
F. The slave has the data for the third read, so it drives DVA on SResp and the data on SData.

G. The master captures the data for the third read from SData. The request-to-response latency for this transfer is 2.

Response Accept

Figure 22 shows examples of the response accept extension used with two read transfers. An additional field, MRespAccept, is added to the response phase. This signal may be used by the master to flow-control the response phase.

Figure 22 Response Accept



Sequence

A. The master starts a read request by driving RD on MCmd and a valid address on MAddr. The slave asserts SCmdAccept immediately, and it drives DVA on SResp and the read data on SData as soon as it sees the read request. The master is not ready to receive the response for the request it just issued, so it deasserts MRespAccept.

B. Since SCmdAccept is asserted, the request phase ends. The master continues to deassert MRespAccept, however. The slave holds SResp and SData steady.

C. The master starts a second read request and is ready for the response from its first request, so it asserts MRespAccept. This corresponds to a response accept latency of 2.

D. Since SCmdAccept is asserted, the request phase ends. The master captures the data for the first read from the slave. Since MRespAccept is asserted, the response phase ends. The slave is not ready to respond to the second read, so it drives NULL on SResp.

E. The slave responds to the second read by driving DVA on SResp and the read data on SData. The master is not ready for the response, however, so it deasserts RespAccept.

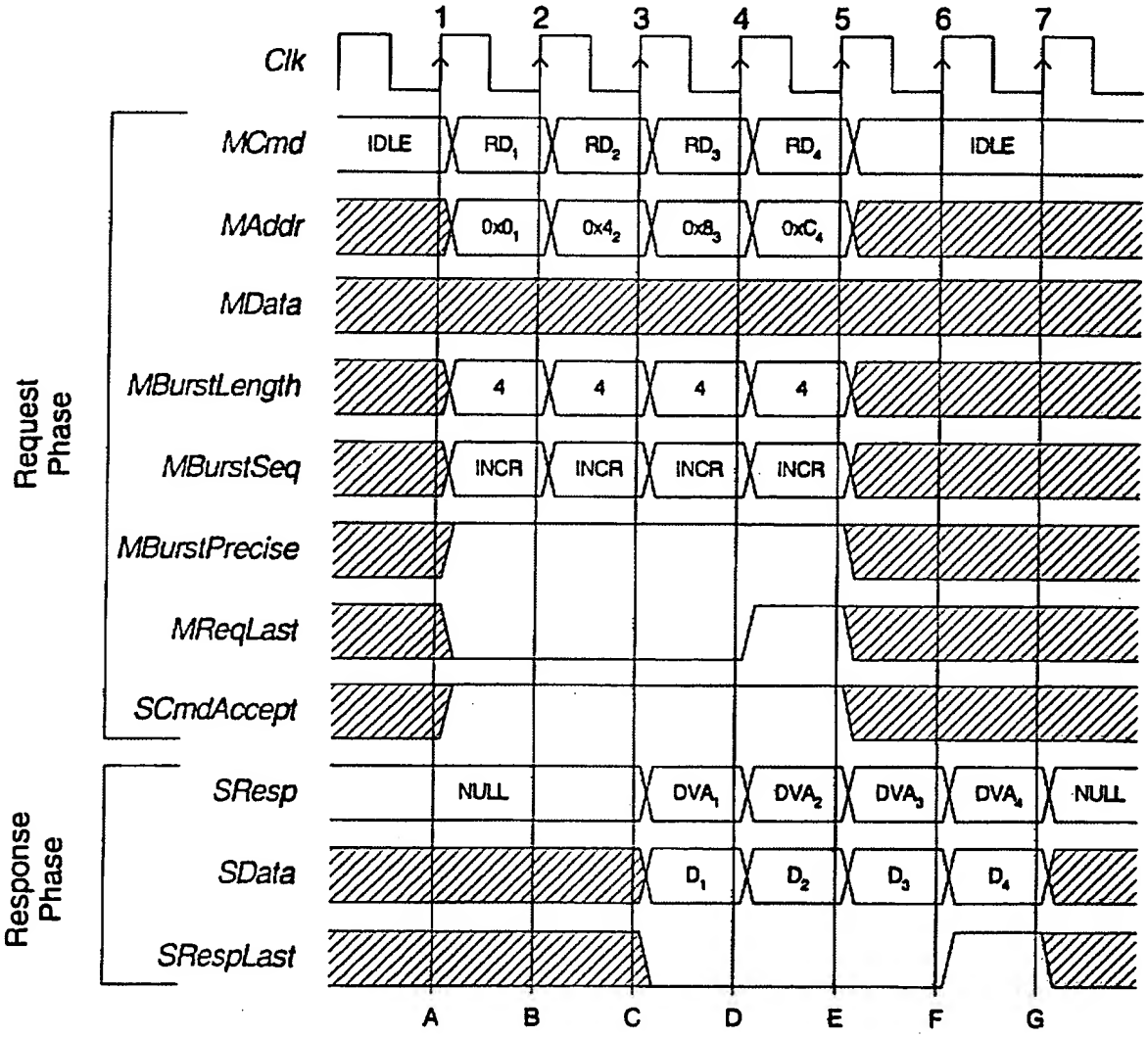
F. The master asserts MRespAccept, for a response accept latency of 1.

G. The master captures the data for the second read from the slave. Since MRespAccept is asserted, the response phase ends.

Incrementing Precise Burst Read

Figure 23 illustrates a burst of four 32-bit words, incrementing precise burst read, with burst framing information (MReqLast/SRespLast). Since the burst is precise, the MBurstLength signal is constant during the whole burst. MReqLast flags the last request of the burst, and SRespLast flags the last response of the burst.

Figure 23 Incrementing Precise Burst Read



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request and keeps SCmdAccept asserted.

C. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the second request and keeps SCmdAccept asserted. The slave responds to the first read by driving DVA on SResp and the read data on SData.

D. The master issues the last request of the burst, incrementing MAddr and asserting MReqLast. The master also captures the data for the first read from the slave. The slave responds to the second request, and captures the address of the third request.

E. The master captures the data for the second read from the slave. The slave responds to the third request and captures the address of the fourth.

F. The master captures the data for the third read from the slave. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

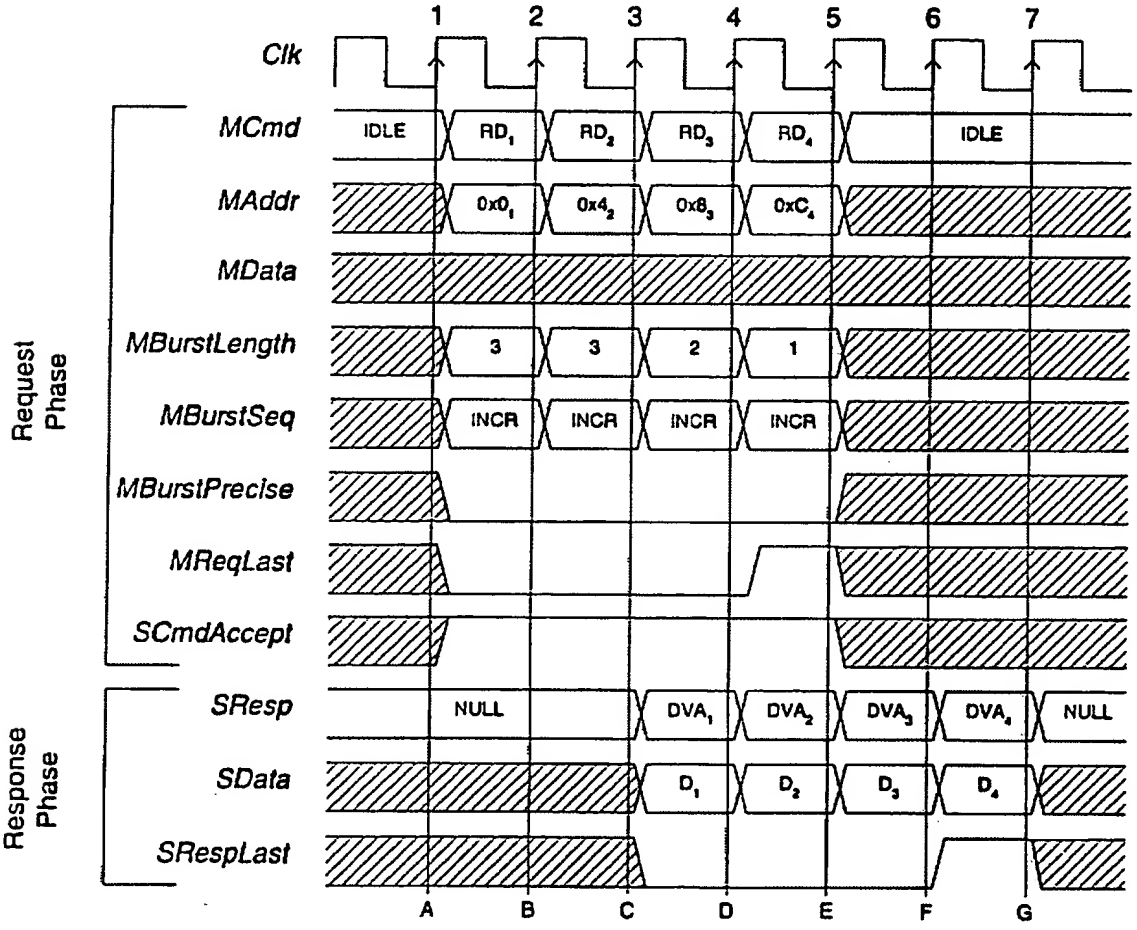
G. The master captures the data for the last read from the slave, ending the last response phase.

Incrementing Imprecise Burst Read

Figure 24 illustrates a burst of four 32-bit words, incrementing imprecise burst read, with burst framing information (MReqLast/SRespLast). MReqLast flags the last request of the burst and SRespLast flags the last response of the burst. The last burst request is signaled primarily by driving the value 1 on MBurstLength.

The burst length sequence (3,3,2,1) is chosen arbitrarily for illustration purposes. The protocol requires that the burst length of the last transfer of the burst be equal to one.

Figure 24 Incrementing Imprecise Burst Read



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, three on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. The burst length is the best guess of the master at this point. MBurstSeq and MBurstPrecise are kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The MBurstLength is set to three, since the master knows the burst is longer than it originally thought. The slave captures the address of the first request and keeps SCmdAccept asserted.

C. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The MBurstLength is set to two. The slave captures the address of the second request, and keeps SCmdAccept asserted. The slave responds to the first read by driving DVA on SResp and the read data on SData.

D. The master issues the last request of the burst, incrementing MAddr, setting MBurstLength to one, and asserting MReqLast. The master also captures the data for the first read from the slave. The slave responds to the second request and captures the address of the last request.

E. The master captures the data for the second read from the slave. The slave responds to the third request.

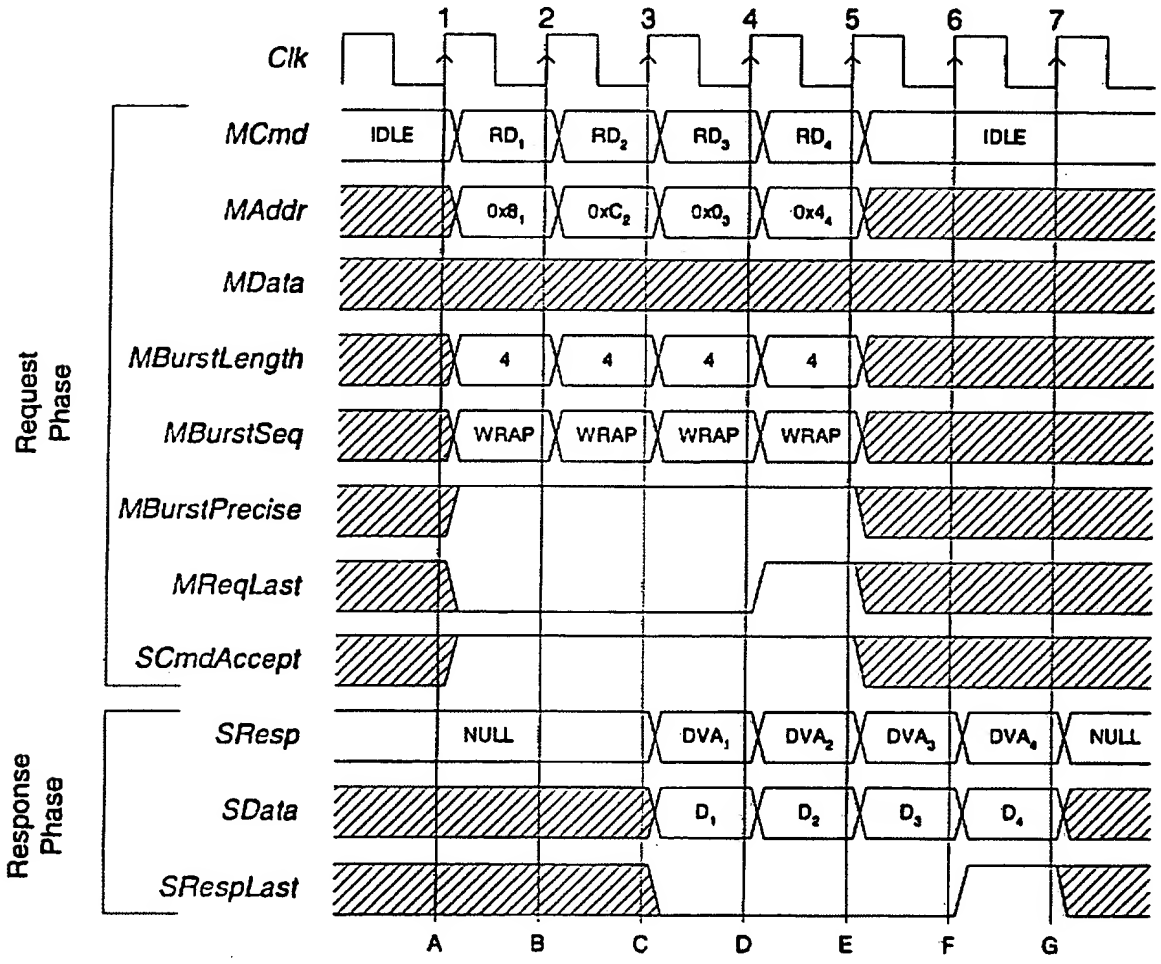
F. The master captures the data for the third read from the slave. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

Wrapping Burst Read

Figure 25 illustrates a burst of four 32-bit words, wrapping burst read, with optional burst framing information (MReqLast/SRespLast). MReqLast flags the last request of the burst and SRespLast flags the last response of the burst. As a wrapping burst is precise, the MBurstLength signal is constant during the whole burst, and must be power of two. The wrapping burst address must be aligned to boundary MBurstLength times the OCP word size in bytes.

Figure 25 Wrapping Burst Read



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, WRAP on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq, and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request, and keeps SCmdAccept asserted.

C. If incremented, the next address would be 0x10. Since the first transfer was from address 0x8 and the burst length is 4, the addresses must be between 0 and 0xF. The master wraps the MAddr to 0, which is the closest alignment boundary. (If the first address were 0x14, the address would wrap to 0x10, rather than 0x20.) The slave captures the address of the second request, and keeps SCmdAccept asserted. The slave responds to the first read by driving DVA on SResp and valid data on SData.

D. The master issues the last request of the burst, incrementing MAddr and asserting MReqLast. The master also captures the data for the first read from the slave. The slave responds to the second request and captures the address of the third.

E. The master captures the data for the second read from the slave. The slave responds to the third request and captures the address of the fourth.

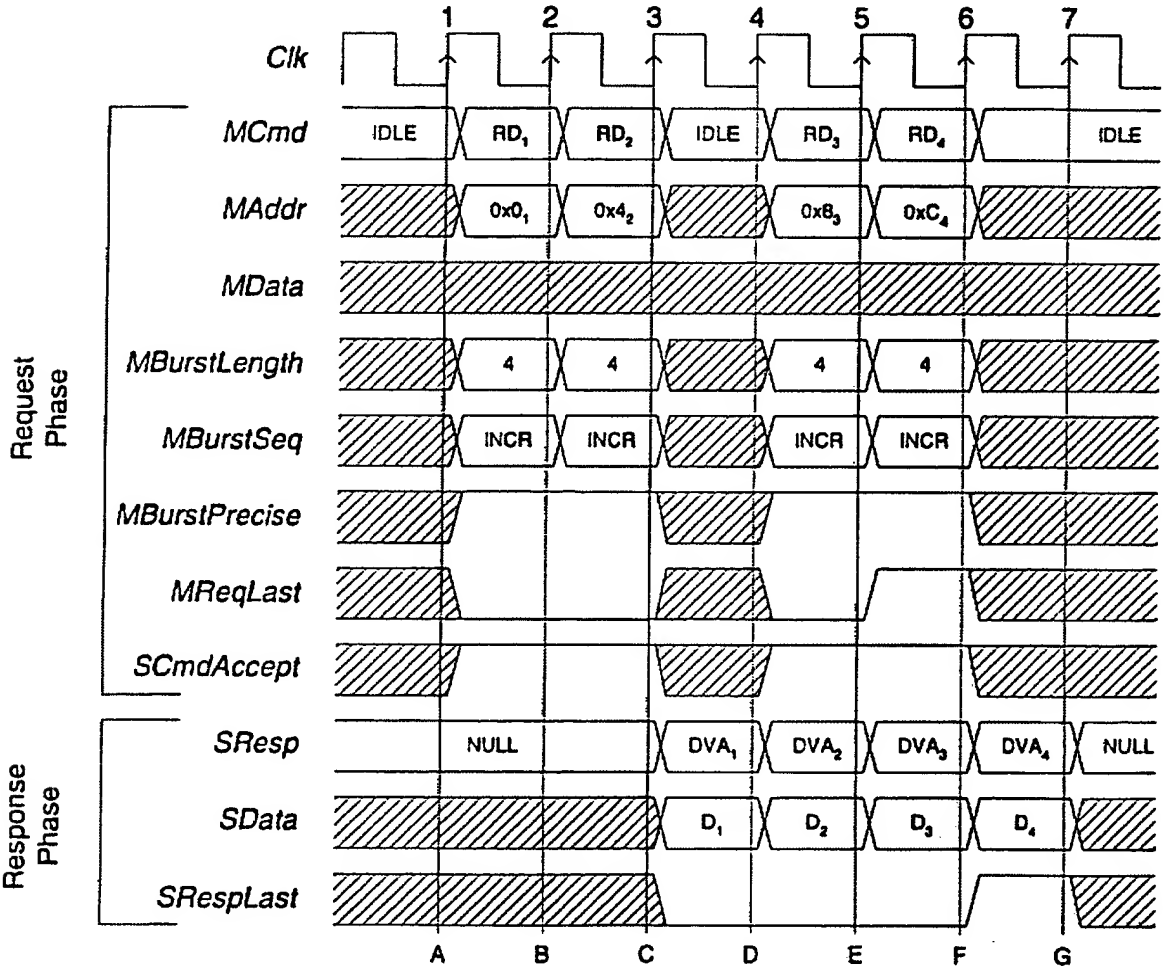
F. The master captures the data for the third read from the slave. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

Incrementing Burst Read with IDLE Request Cycle

Figure 26 illustrates a burst of four 32-bit words, incrementing precise burst read, with an IDLE cycle inserted in the middle. The master may insert IDLE requests in any burst type.

Figure 26 Incrementing Precise Burst Read with IDLE Cycle



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq, and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request and keeps SCmdAccept asserted.

C. The master inserts an IDLE request in the middle of the burst. The slave does not have to deassert SCmdAccept, anticipating more burst requests to come. The slave captures the address of the second request. The slave responds to the first read by driving DVA on SResp and the read data on SData. The slave must keep SRespLast deasserted until the last response.

D. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The master also captures the data for the first read from the slave. The slave responds to the second read by driving DVA on SResp and the read data on SData. If it has the data available for response, the slave does not have to insert a NULL response cycle.

E. The master issues the last request of the burst, incrementing MAddr, and asserting MReqLast. The master also captures the data for the second read from the slave. The slave captures the address of the third request and responds to the third request.

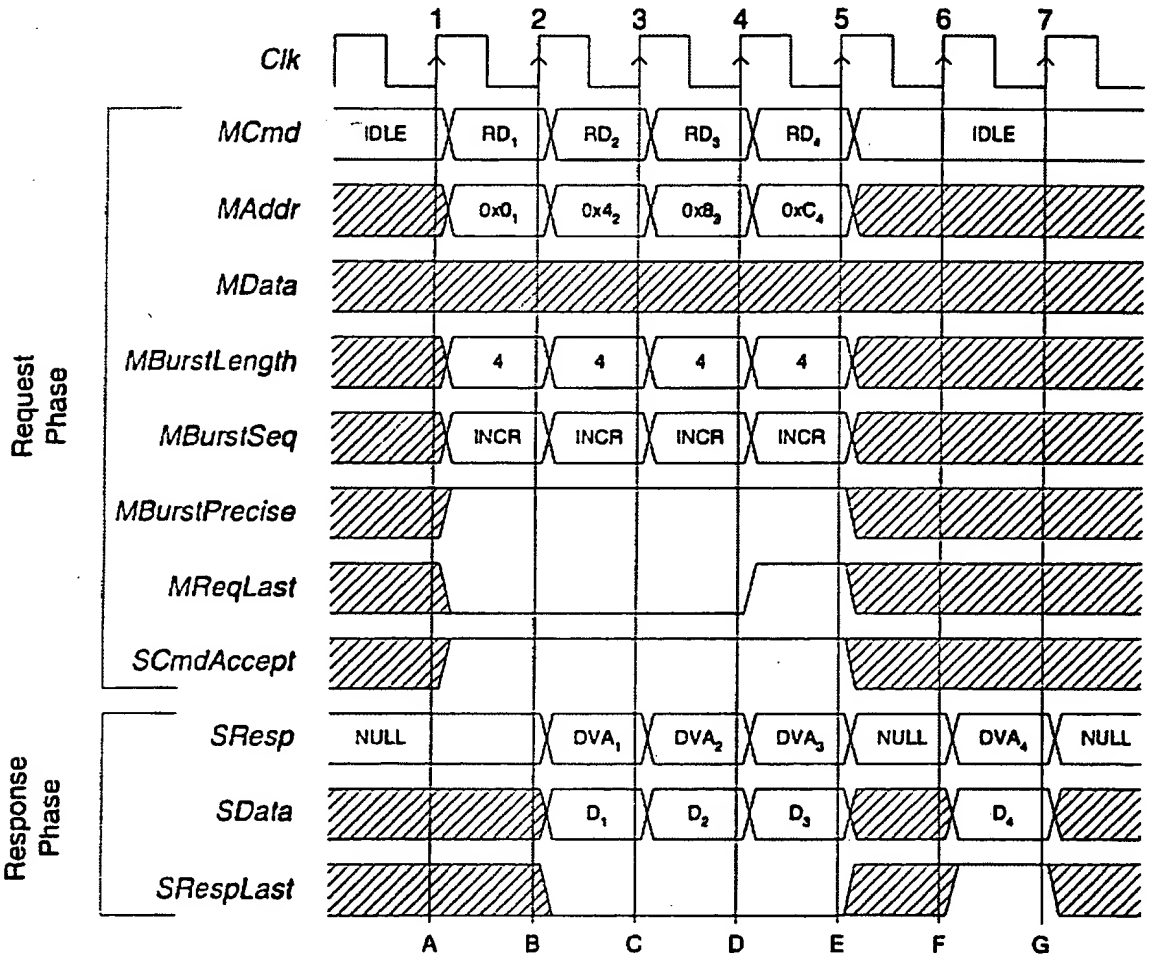
F. The master captures the data for the third read from the slave. The slave captures the address of the fourth request. The slave responds to the fourth request, and asserts SRespLast to indicate the end of the slave burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

Incrementing Burst Read with NULL Response Cycle

Figure 27 illustrates a burst of four 32-bit words, incrementing precise burst read, with a NULL response cycle (wait state) inserted by the slave. Null cycles can be inserted into any burst type by the slave.

Figure 27 Incrementing Burst Read with Null Cycle



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request and keeps SCmdAccept asserted. The slave responds to the first request by driving DVA on SResp and the read data on SData. The slave must keep SRespLast deasserted until the last response.

C. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The master also captures the data for the first read from the slave. The slave captures the address of the second request and keeps SCmdAccept asserted. The slave responds to the second request.

D. The master issues the last request of the burst, incrementing MAddr and asserting MReqLast. The master also captures the data for the second read from the slave. The slave captures the address of the third request and keeps SCmdAccept asserted. The slave responds to the third request.

E. The master captures the data for the third read from the slave. The slave captures the address of the fourth request and keeps SCmdAccept asserted. The slave cannot respond to the fourth request, so it inserts NULL to SResp.

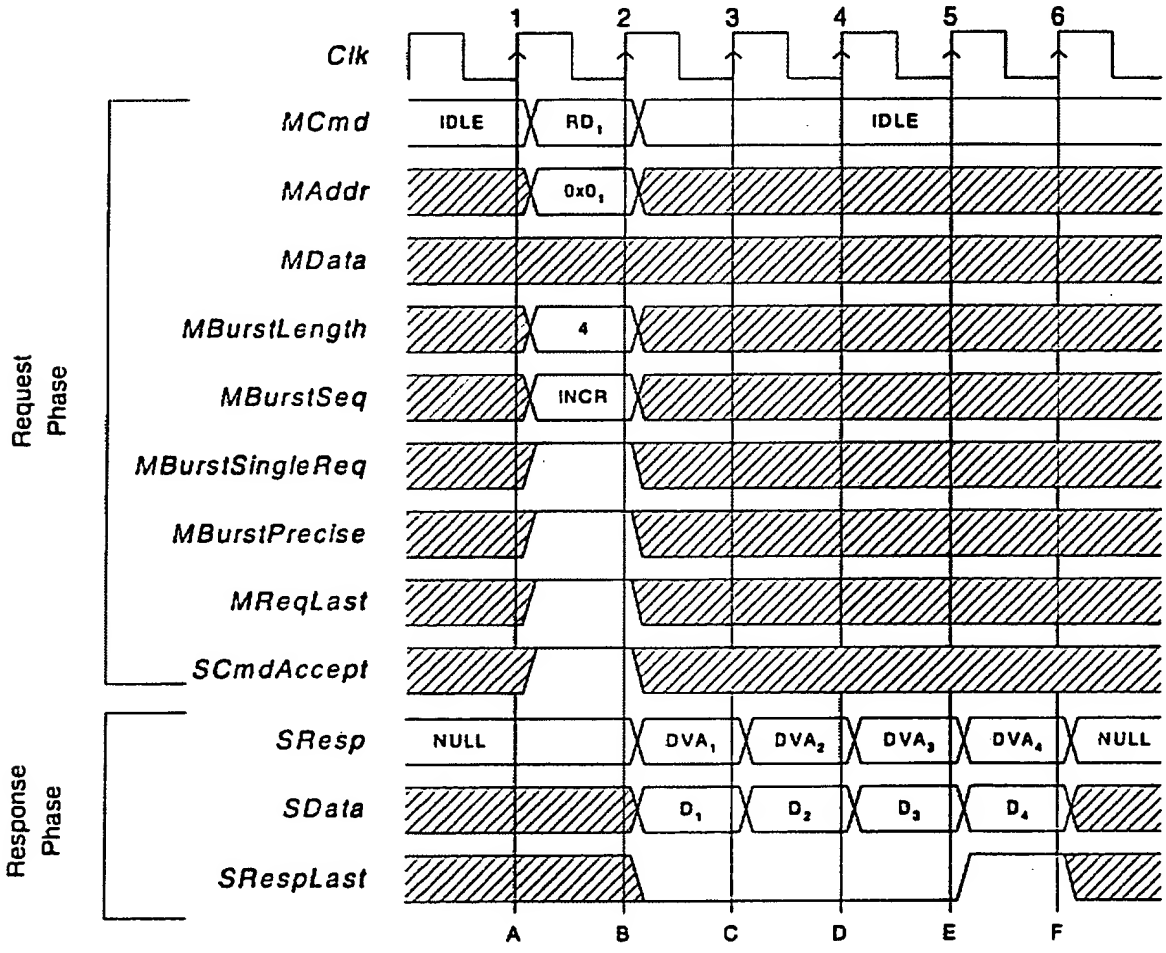
F. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

Single Request Burst Read

Figure 28 illustrates a single request, multiple data burst read. The master provides the burst length, start address, and burst sequence, and identifies the burst as a single request with the MBurstSingleReq signal. A single request burst is always precise.

Figure 28 Single Request Burst Read



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise, and MBurstSingleReq. The MBurstPrecise and MBurstSingleReq signals would normally be tied off to logic 1, which is not supplied by the master. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master completes the request cycles. The slave captures the address of the request. The slave responds to the request by driving DVA on SResp and the first response data on SData. The slave must keep SRespLast deasserted until the last response.

C. The master captures the first response data. The slave issues the second response.

D. The master captures the second response data. The slave issues the third response.

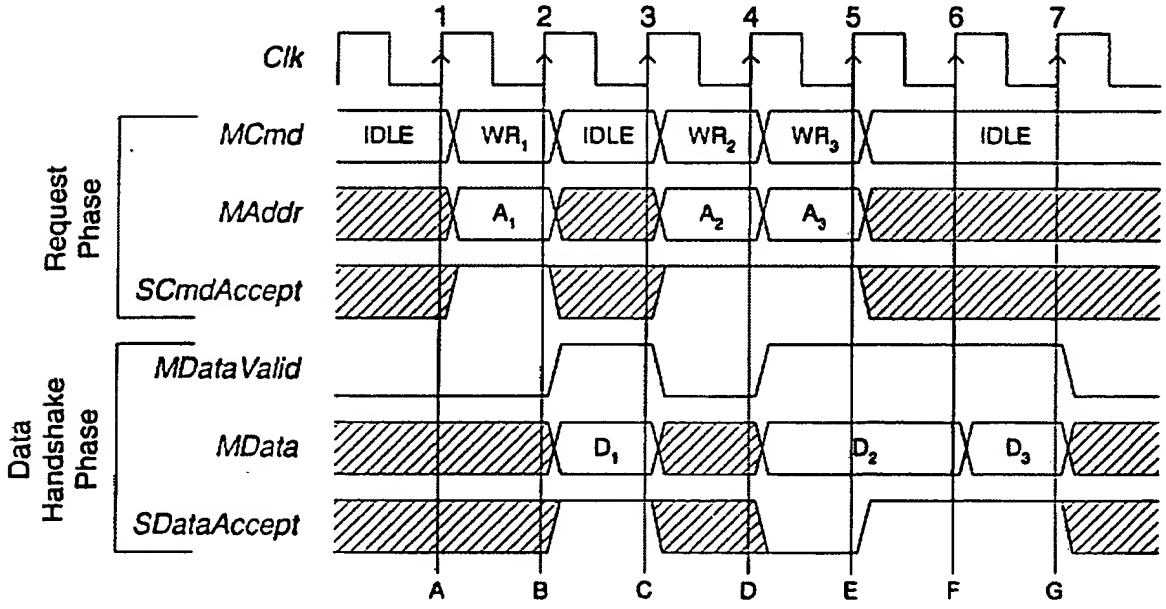
E. The master captures the third response data. The slave issues the fourth response, and asserts SRespLast to indicate the last response of the burst.

F. The master captures the last response data.

Datahandshake Extension

Figure 29 shows three writes with no responses using the datahandshake extension. This extension adds the datahandshake phase, which is completely independent of the request and response phases. Two signals, MDataValid and SDataAccept, are added, and MData is moved from the request phase to the datahandshake phase.

Figure 29 Datahandshake Extension



Sequence

A. The master starts a write request by driving WR on MCmd and a valid address on MAddr. It does not yet have the write data, however, so it deasserts MDataValid. The slave asserts SCmdAccept. It does not need to assert or deassert SDataAccept yet, because MDataValid is still deasserted.

B. The slave captures the write address from the master. The master is now ready to transfer the write data, so it asserts MDataValid and drives the data on MData, starting the datahandshake phase. The slave is ready to accept the data immediately, so it asserts SDataAccept. This corresponds to a data accept latency of 0.

C. The master deasserts MDataValid since it has no more data to transfer. (Like MCmd and SResp, MDataValid must always be in a valid, specified state.) The slave captures the write data from MData, completing the transfer. The master starts a second write request by driving WR on MCmd and a valid address on MAddr.

D. Since SCmdAccept is asserted, the master immediately starts a third write request. It also asserts MDataValid and presents the write data of the second write on MData. The slave is not ready for the data yet, so it deasserts SDataAccept.

E. The master sees that SDataAccept is deasserted, so it holds the values of MDataValid and MData. The slave asserts SDataAccept, for a data accept latency of 1.

F. Since SDataAccept is asserted, the datahandshake phase ends. The master is ready to deliver the write data for the third request, so it keeps MDataValid asserted and presents the data on MData. The slave captures the data for the second write from MData, and keeps SDataAccept asserted, for a data accept latency of 0 for the third write.

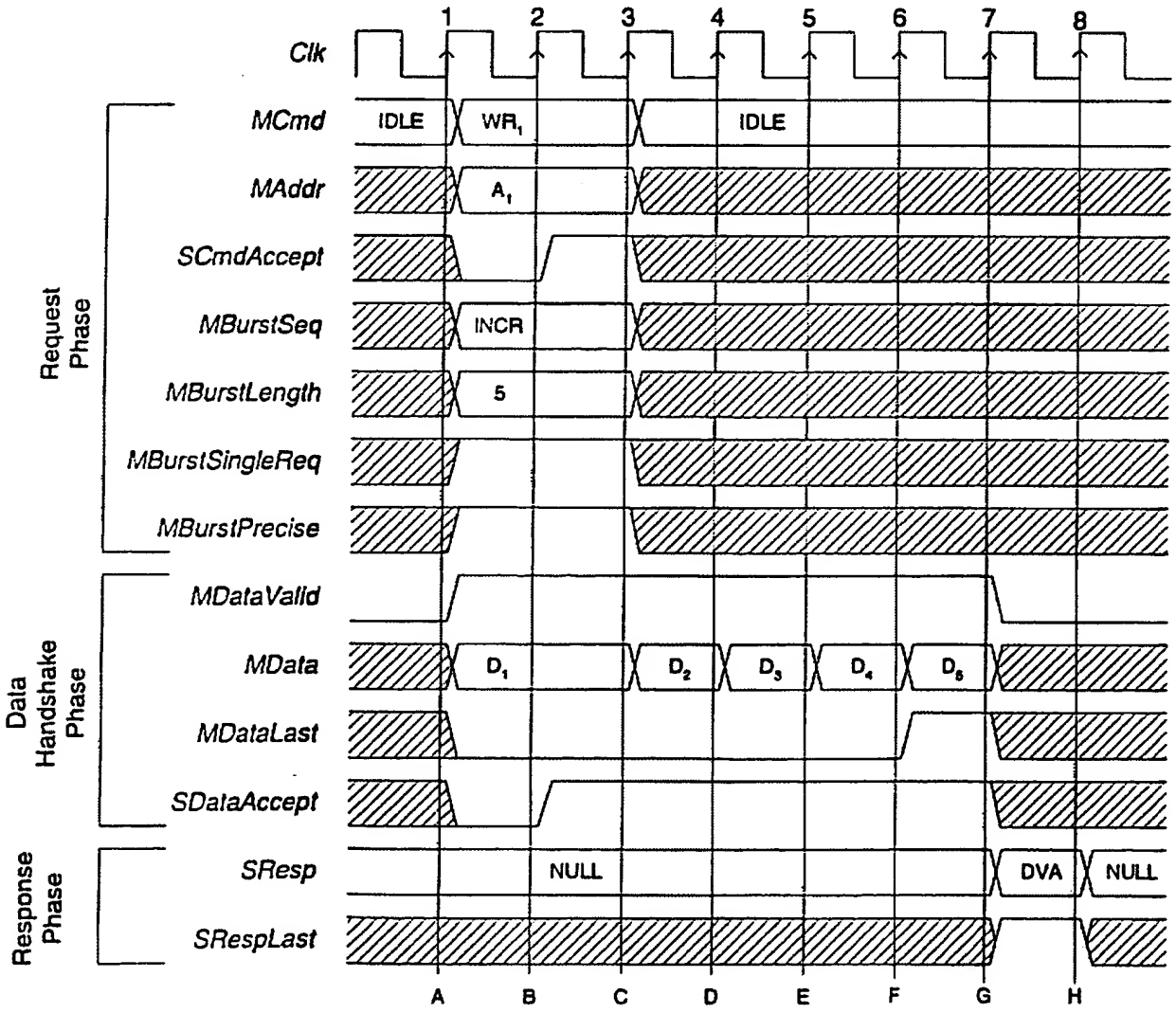
G. Since SDataAccept is asserted, the datahandshake phase ends. The slave captures the data for the third write from MData.

Burst Write with Combined Request and Data

Figure 30 illustrates a single request, multiple data burst write, with datahandshake signaling. Through the request handshake, the master provides the burst length, the start address, and burst sequence, and identifies the burst as a single request with the MBurstSingleReq signal.

The write data is transferred with a datahandshake extension (see Figure 29). The parameter `reqdata_together` forces the first data phase to start with the request, making the design of a slave state machine easier, since it only needs to track one request handshake during the burst. Without this parameter, the MDataValid signal could be asserted any time after the first request. If datahandshake is not used, a single-request write burst is not possible; instead a request is required for each burst word.

Figure 30 Burst Write with Combined Request and Data



Sequence

A. The master starts a write request by driving WR on MCmd, a valid address on MAddr, INCR on MBurstSeq, five on MBurstLength, and asserts the MBurstPrecise and MBurstSingleReq signals. The master also asserts the MDataValid signal, drives valid data on MData, and deasserts MDataLast. The MDataLast signal must remain deasserted until the last data cycle.

B. Since it has not received SCmdAccept or SDataAccept, the master holds the request phase signals, keeps MDataValid asserted, and MData steady. The slave asserts SCmdAccept and SDataAccept to indicate it is ready to accept the request and the first data phase.

C. The master completes the request phase, asserts MDataValid and drives new data to MData. The slave captures the initial data and keeps SDataAccept asserted to indicate it is ready to accept more data.

D. The master asserts MDataValid and drives new data to MData. The slave captures the second data phase and keeps SDataAccept asserted to indicate it is ready to accept more data.

E. The master asserts MDataValid and drives new data to MData. The slave captures the third data phase and keeps SDataAccept asserted to indicate it is ready to accept more data.

F. The master asserts MDataValid, drives new data to MData, and asserts MDataLast to identify the last data in the burst. The slave captures the fourth data phase and keeps SDataAccept asserted to indicate it is ready to accept more data.

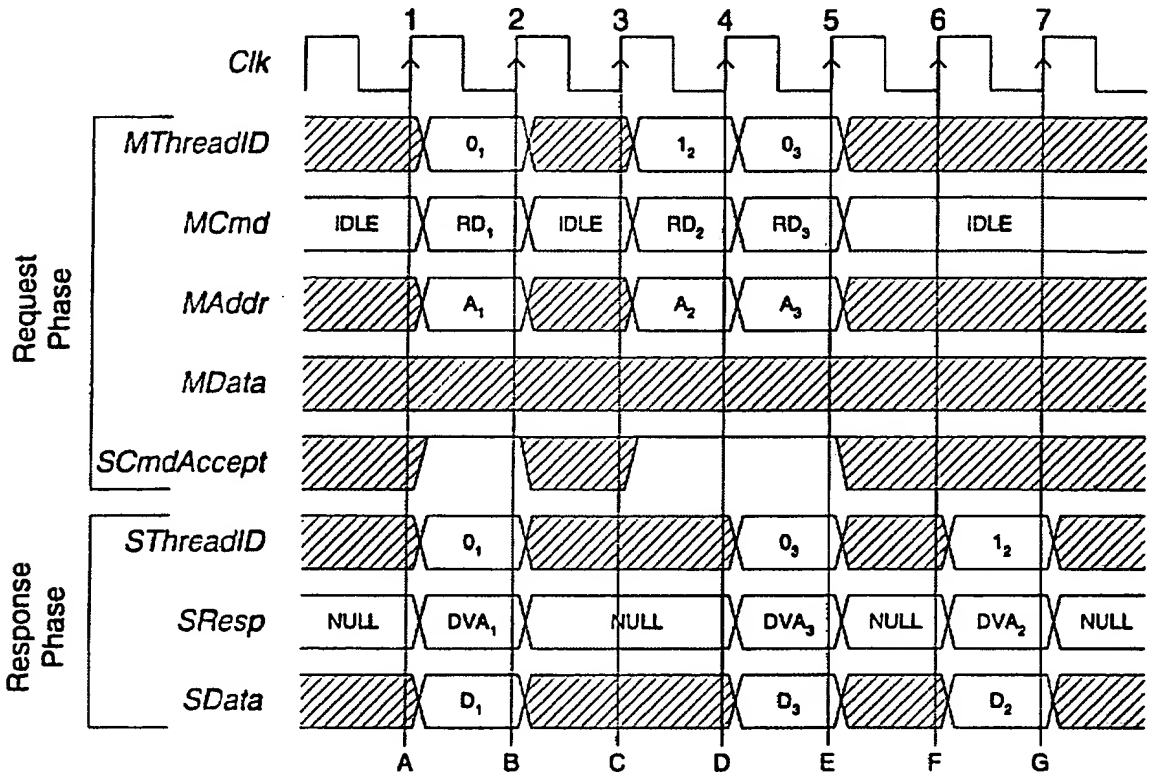
G. The slave captures the last data phase and address.

This example also shows how the slave issues SResp at the end of a burst (when the optional write response is configured in the interface). For single request / multiple data bursts there is only a single response, and it can be issued after the last data has been detected by the slave. The SResp is NULL until point G. in the diagram. The slave may use code DVA to indicate a successful burst, or ERR for an unsuccessful one.

Threaded Read

Figure 31 illustrates out-of-order completion of read transfers using the OCP thread extension. This diagram is developed from Figure 21 on page 105. The thread IDs, MThreadID and SThreadID, have been added, and the order of two of the responses has been changed.

Figure 31 Threaded Read



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The master also drives a 0 on MThreadID, indicating that this read request is for thread 0. The slave asserts SCmdAccept, for a request accept latency of 0. When the slave sees the read command, it responds with DVA on SResp and valid data on SData. The slave also drives a 0 on SThreadID, indicating that this response is for thread 0.

B. Since SCmdAccept is asserted, the request phase ends. The master sees that SResp is DVA and captures the read data from SData. Because the request was accepted and the response was presented in the same cycle, the request-to-response latency is 0.

C. The master launches a new read request, but this time it is for thread 1. The slave asserts SCmdAccept, however, it is not ready to respond.

D. Since SCmdAccept is asserted, the master can launch another read request. This request is for thread 0, so MThreadID is switched back to 0. The slave captures the address of the second read for thread 1, but it begins driving DVA on SResp, data on SData, and a 0 on SThreadID. This means that it is responding to the third read, before the second read.

E. Since SCmdAccept is asserted, the third request ends. The master sees that the slave has produced a valid response to the third read and captures the data from SData. The request-to-response latency for this transfer is 0.

F. The slave has the data for the second read, so it drives DVA on SResp, data on SData, and a 1 on SThreadID.

G. The master captures the data for the second read from SData. The request-to-response latency for this transfer is 3.

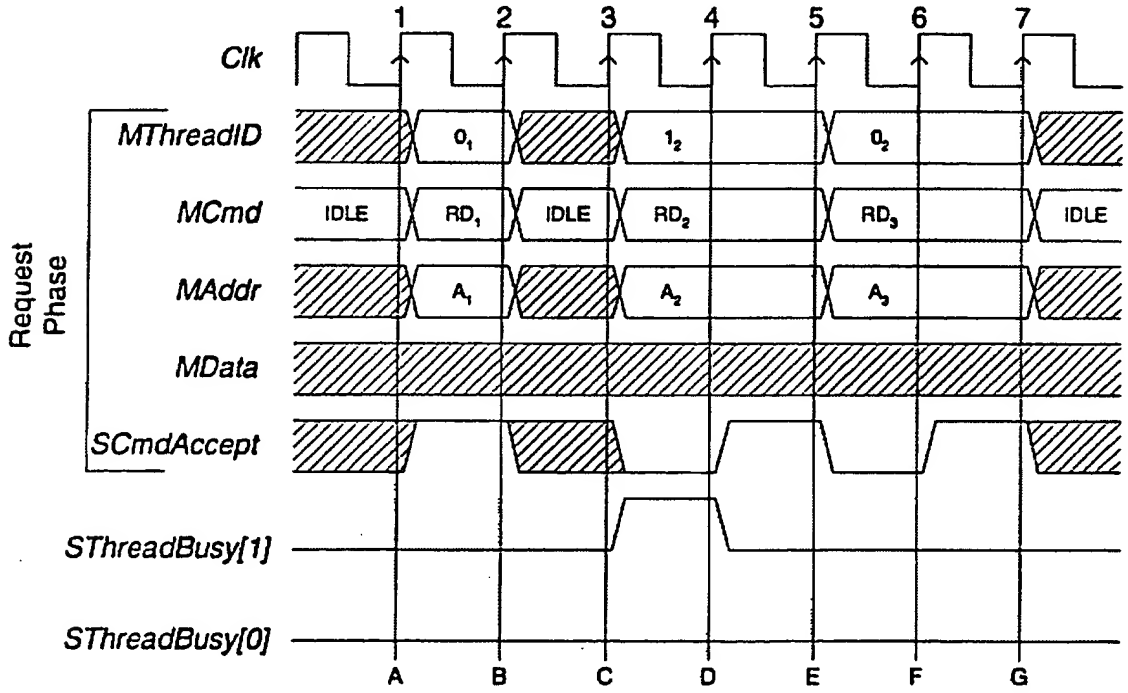
Threaded Read with Thread Busy

Figure 32 illustrates the out-of-order completion of read transfers using the OCP thread extension. The change to Figure 31 is the addition of thread busy signals. In this example, the thread busy is only a hint, since the `sthreadbusy_exact` parameter is not set. In this case the master may ignore the SThreadBusy signals, and the slave does not have to accept requests even when it is not busy.

When thread busy is treated as a hint and a request or thread is not accepted, the interface may block for all threads. Blocking of this type can be avoided by treating thread busy as an exact signal using the `sthreadbusy_exact` parameter. For an example, see "Threaded Read with Thread Busy Exact".

This example shows only the request part of the read transfers. The response part can use a similar mechanism for thread busy.

Figure 32 Threaded Read with Thread Busy



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The master also drives a 0 on MThreadID, associating this read request with thread 0. The slave asserts SCmdAccept for a request accept latency of 0.

B. Since SCmdAccept is asserted, the request phase ends.

C. The slave asserts SThreadBusy[1] since it is not ready to accept requests on thread 1. The master ignores the hint, and launches a new read request for thread 1. The master can issue a request even though the slave asserts SThreadbusy (see transfer 2). All threads are now blocked.

D. The slave deasserts SThreadBusy[1] and asserts SCmdAccept to complete the request for thread 1.

E. Since SCmdAccept is asserted, the second request ends. The master issues a new request to thread 0. The slave is not ready to accept the request, and indicates this condition by keeping SCmdAccept deasserted. It chooses not to assert SThreadBusy[0]. The slave does not have to assert SCmdAccept for a request, even if it did not assert SThreadbusy (see transfer 3).

F. The slave asserts the SCmdAccept to complete the request on thread 0.

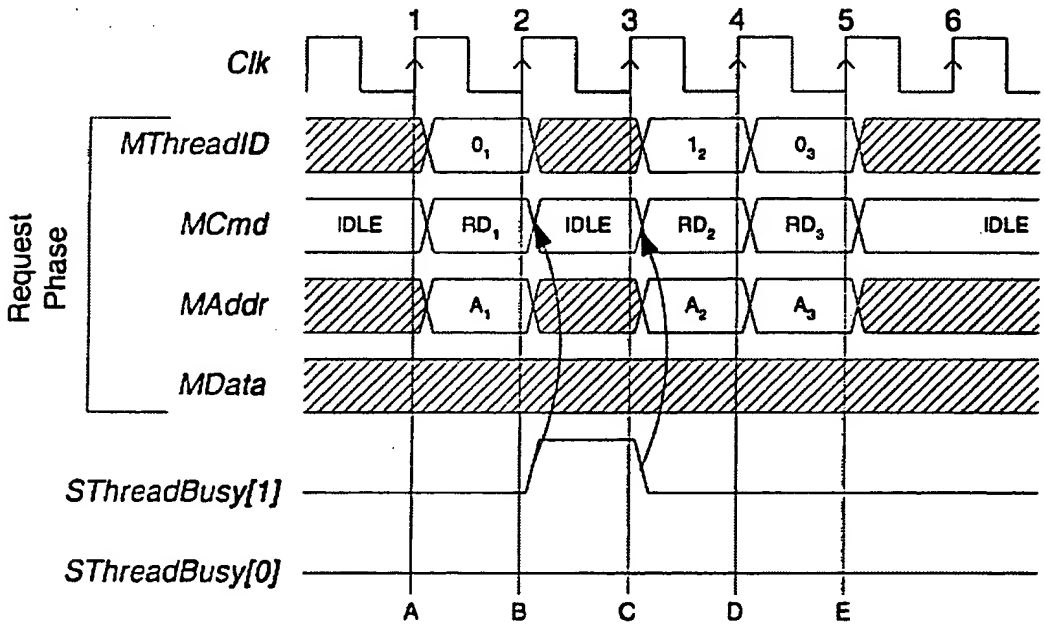
G. The master captures the SCmdAccept to complete the requests.

Threaded Read with Thread Busy Exact

Figure 33 illustrates the out-of-order completion of read transfers using the OCP thread extension. Because the sthreadbusy_exact parameter is set, the master may not ignore the SThreadBusy signals. The master is using SThreadBusy to control thread arbitration, so it cannot present a command on Thread 1 as the slave asserts SThreadbusy[1].

This example shows only the request part of the read transfers. The response part can use a similar mechanism for thread busy.

Figure 33 Threaded Read with Thread Busy Exact



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The master also drives a 0 on MThreadID, indicating that this read request is for thread 0.

B. Since SThreadBusy[0] is not asserted, the request phase ends. The slave samples the data and address and asserts SThreadBusy[1] since it is not ready to accept requests on thread 1. The master is prevented from sending a request on thread 1, though it can send a request on another thread.

C. The slave deasserts SThreadBusy[1] and the master can send the request on thread 1.

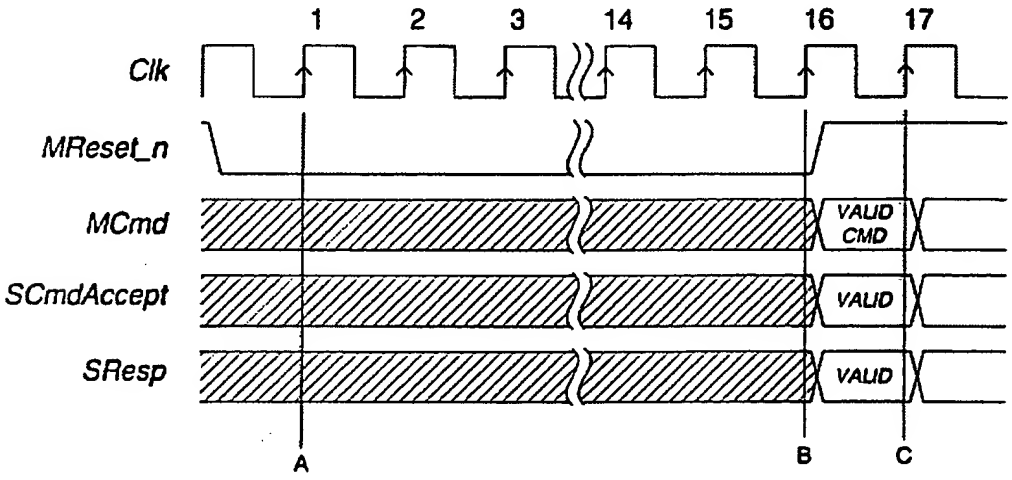
D. Since SThreadBusy[1] is not asserted, the request phase ends and the slave must sample the data and address. The master can send a request on thread 0 (or thread 1).

E. Since SThreadBusy[0] is not asserted, the request phase ends and the slave must sample the data and address.

Reset

Figure 34 shows the timing of the reset sequence with MReset_n driven from the master to the slave. MReset_n must be asserted for at least 16 cycles of Clk to ensure that the master and slave reach a consistent internal state.

Figure 34 Reset Sequence



Sequence

A. MReset_n is sampled active on this clock edge. Master and slave now ignore all other OCP signals.

B. MReset_n is asserted for at least 16 Clk cycles.

C. A new transfer may begin on the same clock edge that MReset_n is sampled deasserted.

9 Developers Guidelines

This chapter collects examples and implementation tips that can help you make effective use of the Open Core Protocol and does not provide any additional specification material. This chapter groups together a variety of topics including discussions of:

1. The basic OCP with an emphasis on signal timing, state machines and OCP subsets.
2. Simple extensions that cover byte enables, multiple address spaces and in-band information
3. An overview of the new 2.0 burst capabilities
4. The concepts of threading and connections
5. OCP features addressing the new write semantics, synchronization issues, and endianness
6. Sideband signals with an emphasis on reset management
7. A description of the debug and test interface

Basic OCP

This section considers the different OCP phases, their relationships, and identifies sensitive timing-related areas. The section includes descriptions of OCP compliant state machines, and also discusses the OCP parameters needed to define simple OCP interfaces.

Signal Timing

The Open Core Protocol data transfer model allows many different types of existing legacy IP cores to be bridged to the OCP without adding expensive glue logic structures that include address or data storage. As such, it is possible to draw many state machine diagrams that are compliant with the OCP protocol. This section describes some common state machine models that can be used with the OCP, together with guidance on the use of those models.

Two-way handshaking is the general principle of the dataflow signals in the OCP interface. A group of signals is asserted and must be held steady until the corresponding accept signal is asserted. This allows the receiver of a signal to force the sender to hold the signals steady until it has completely processed them. This principle produces implementations with fewer latches for temporary storage.

OCP principles are built around three fundamental decoupled phases: the request phase, the response phase, and the datahandshake phase.

Request Phase

Request flow control relies on standard request/accept handshaking signals: MCmd and SCmdAccept. Note that in version 2.0 of this specification, SCmdAccept becomes an optional signal, enabled by the cmdaccept parameter. When the signal is not physically present on the interface, it naturally defaults to 1, meaning that a request phase in that case lasts exactly one clock cycle.

The request phase begins when the master drives MCmd to a value other than Idle. When MCmd != Idle, MCmd is referred to as asserted. All of the other request phase outputs of the master must become valid during the same clock cycle as MCmd asserted, and be held steady until the request phase ends. The request phase ends when SCmdAccept is sampled asserted (true) by the rising edge of Clk. The slave can assert SCmdAccept in the same cycle that MCmd is asserted, or stay negated for several Clk cycles. The latter choice allows the slave to force the master to hold its request phase outputs until the slave can accomplish its access without latching address or data signals.

The slave designer chooses the delay between MCmd asserted and SCmdAccept asserted based on the desired area, timing, and throughput characteristics of the slave.

As the request phase does not begin until MCmd is asserted, SCmdAccept is a "don't care" while MCmd is not asserted so SCmdAccept can be asserted before MCmd. This allows some area-sensitive, low frequency slaves to tie SCmdAccept asserted, as long as they can always complete their transfer responsibilities in the same cycle that MCmd is asserted. Since an MCmd value of Idle specifies the absence of a valid command, the master can assert MCmd independently of the current setting of SCmdAccept.

The highest throughput that can be achieved with the OCP is one data transfer per Clk cycle. High-throughput slaves can approach this rate by providing sufficient internal resources to end most request phases in the same Clk cycle that they start. This implies a combinational path from the

master's MCmd output into slave logic, then back out the slaves SCmdAccept output and back into a state machine in the master. If the master has additional requests to present, it can start a new request phase on the next Clk cycle. Achieving such high throughput in high-frequency systems requires careful design including cycle time budgeting as described in "Level2 Timing" on page 162.

Response Phase

The response phase begins when the slave drives SResp to a value other than NULL. When SResp != NULL, SResp is referred to as asserted. All of the other response phase outputs of the slave must become valid during the same Clk cycle as SResp asserted, and be held steady until the response phase ends. The response phase ends when MRespAccept is sampled asserted (true) by the rising edge of Clk; if MRespAccept is not configured into a particular OCP, MRespAccept is assumed to be always asserted (that is, the response phase always ends in the same cycle it begins). If present, the master can assert MRespAccept in the same cycle that MResp is asserted, or it may stay negated for several Clk cycles. The latter choice allows the master to force the slave to hold its response phase outputs so the master can finish the transfer without latching the data signals.

Since the response phase does not begin until SResp is asserted, MRespAccept is a "don't care" while SResp is not asserted so MRespAccept can be asserted before SResp. Since an SResp value of NULL specifies the absence of a valid response, the slave can assert SResp independently of the current setting of MRespAccept.

In high-throughput systems, careful use of MRespAccept can result in significant area savings. To maintain high throughput, systems traditionally introduce *pipelining*, where later requests begin before earlier requests have finished. Pipelining is particularly important to optimize Read accesses to main memory.

The OCP supports pipelining with its basic request/response protocol, since a master is free to start the second request phase as soon as the first has finished (before the first response phase, in many cases). However, without MRespAccept, the master must have sufficient storage resources to receive all of the data it has requested. This is not an issue for some masters, but can be expensive when the master is part of a bridge between subsystems such as computer buses. While the original system initiator may have enough storage, the intermediate bridge may not. If the slave has storage resources (or the ability to flow control data that it is requesting), then allowing the master to de-assert MRespAccept enables the system to operate at high throughput without duplicating worst-case storage requirements across the die.

If a target core natively includes buffering resources that can be used for response flow control at little cost, using MRespAccept can reduce the response buffering requirement in a complex SOC interconnect.

Most simple or low-throughput slave IP cores need not implement MRespAccept. Misuse of MRespAccept makes the slave's job more difficult, because it adds extra conditions (and states) to the slave's logic.

Datahandshake Phase

The datahandshake extension allows the de-coupling of a write address from write data. The extension is typically only useful for master and slave devices that require the throughput advantages available through transfer pipelining (particularly memory). When the datahandshake phase is not present in a configured OCP, MData becomes a request phase signal.

The datahandshake phase begins when the master asserts MDataValid. The other datahandshake phase outputs of the master must become valid during the same Clk cycle while MDataValid is asserted, and be held steady until the datahandshake phase ends. The datahandshake phase ends when SDataAccept is sampled asserted (true) by the rising edge of Clk. The slave can assert SDataAccept in the same cycle that MDataValid is asserted, or it can stay negated for several Clk cycles. The latter choice allows the slave to force the master to hold its datahandshake phase outputs so the slave can accomplish its access without latching data signals.

The datahandshake phase does not begin until MDataValid is asserted. While MDataValid is not asserted, SDataAccept is a "don't care". SDataAccept can be asserted before MDataValid. Since MDataValid not being asserted specifies the absence of valid data, the master can assert MDataValid independently of the current setting of SDataAccept.

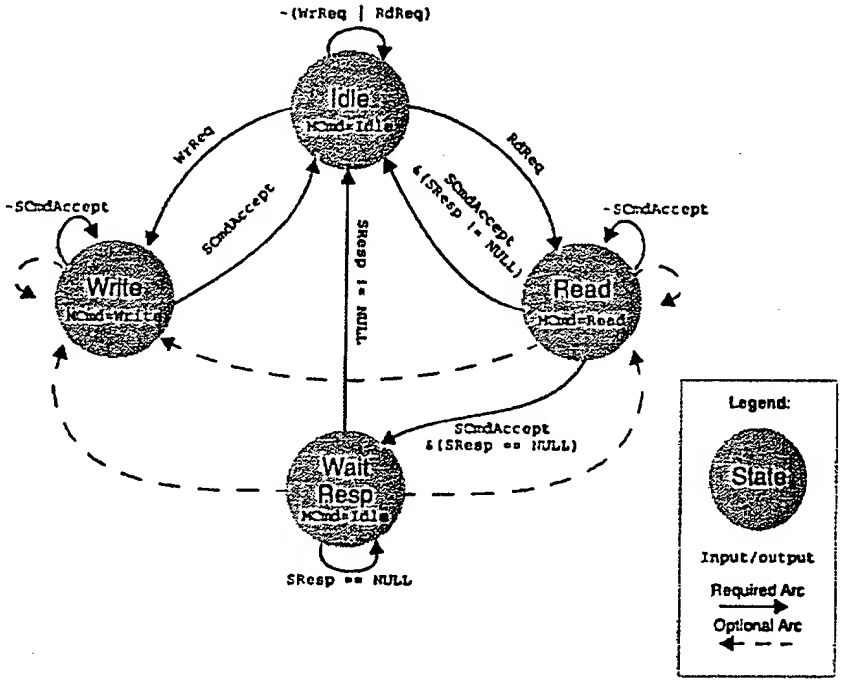
State Machine Examples

The sample state machine implementations in this section use only the features of the basic OCP, request and response phases (the datahandshake phase is not discussed here but can be derived). The examples highlight the flexibility of the basic OCP.

Sequential Master

The first example is a medium-throughput, high-frequency master design. To achieve high frequency, the implementation is a completely sequential (that is, Moore state machine) design. Figure 35 shows the state machine associated with the master's OCP.

Figure 35 Sequential Master



Not shown is the internal circuitry of the master. It is assumed that the master provides the state machine with two control wire inputs, WrReq and RdReq, which ask the state machine to initiate a write transfer and a read transfer, respectively. The state machine indicates back to the master the completion of a transfer as it transitions to its idle state.

Since this is a Moore state machine, the outputs are only a function of the current state. The master cannot begin a request phase by asserting MCmd until it has entered a requesting state (either write or read), based upon the WrReq and RdReq inputs. In the requesting states, the master begins a request phase that continues until the slave asserts SCmdAccept. At this point (this example assumes write posting with no response on writes), a Write command is complete, so the master transitions back to the idle state.

In case of a Read command, the next state is dependent upon whether the slave has begun the response phase or not. Since MRespAccept is not enabled in this example, the response phase always ends in the cycle it begins, so the master may transition back to the idle state if SResp is asserted. If the response phase has not begun, then the next state is wait resp, where the master waits until the response phase begins.

The maximum throughput of this design is one transfer every other cycle, since each transfer ends with at least one cycle of idle. The designer could improve the throughput (given a cooperative slave) by adding the state transitions marked with dashed lines. This would skip the idle state when there are more pending transfers by initiating a new request phase on the cycle after the previous request or response phase. Also, the Moore state machine

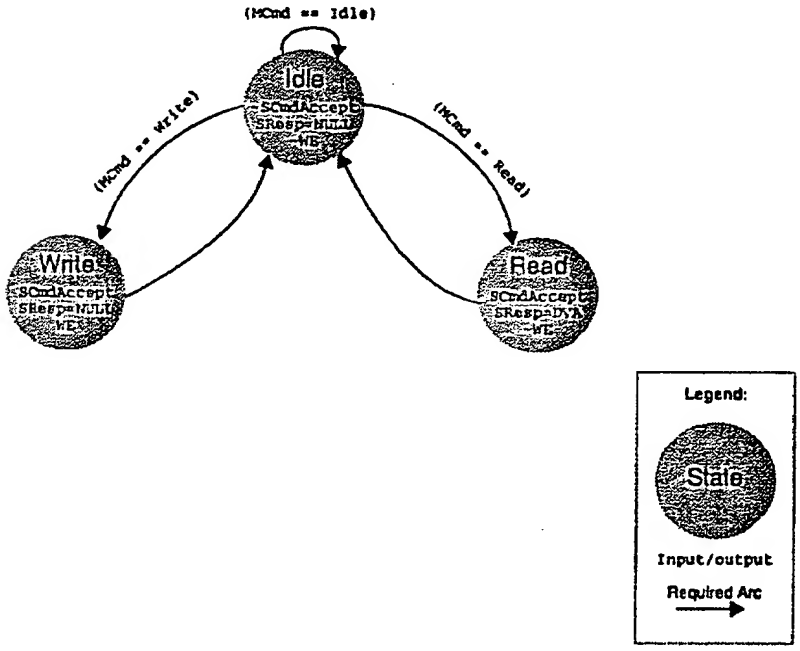
adds up to a cycle of latency onto the idle to request transition, depending on the arrival time of WrReq and RdReq. This cost is addressed in "Combinational Master" on page 135.

The benefits of this design style include very simple timing, since the master request phase outputs deliver a full cycle of setup time, and minimal logic depth associated with SResp.

Sequential Slave

An analogous design point on the slave side is shown in Figure 36. This slave's OCP logic is a Moore state machine. The slave is capable of servicing an OCP read with one Clk cycle latency. On an OCP write, the slave needs the master to hold MData and the associated control fields steady for a complete cycle so the slave's write pulse generator will store the desired data into the desired location. The state machine reacts only to the OCP (the internal operation of the slave never prevents it from servicing a request), and the only non-OCP output of the state machine is the enable (WE) for the write pulse generator.

Figure 36 Sequential OCP Slave



The state machine begins in an idle state, where it de-asserts SCmdAccept and SResp. When it detects the start of a request phase, it transitions to either a read or a write state, based upon MCmd. Since the slave will always complete its task in one cycle, both active states end the request phase (by asserting SCmdAccept), and the read state also begins the response phase. Since MRespAccept is not enabled in this example, the response phase will end in the same cycle it begins. Writes without responses are assumed so SResp is NULL during the write state. Finally, the state machine triggers the write pulse generator in its write state, since the request phase outputs of the master will be held steady until the state machine transitions back to idle.

As is the case for the sequential master shown in Figure 35 on page 133, this state machine limits the maximum throughput of the OCP to one transfer every other cycle. There is no simple way to modify this design to achieve one transfer per cycle, since the underlying slave is only capable of one write every other cycle. With a Moore machine representation, the only way to achieve one transfer per cycle is to assert SCmdAccept unconditionally (since it cannot react to the current request phase signals until the next Clk cycle). Solving this performance issue requires a combinational state machine.

Since the outputs depend upon the state machine, the sequential OCP slave has attractive timing properties. It will operate at very high frequencies (providing the internal logic of the slave can run that quickly).

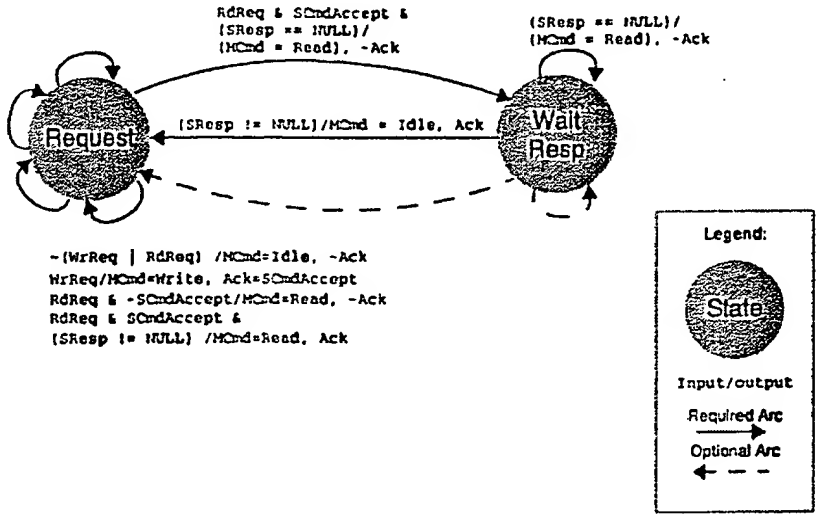
This state machine can be extended to accommodate slaves with internal latency of more than one cycle by adding a counting state between idle and one or both of the active states.

Combinational Master

"Sequential Master" on page 132 describes the transfer latency penalty associated with a Moore state machine implementation of an OCP master. An attractive approach to improving overall performance while reducing circuit area is to consider a combinational Mealy state machine representation. Assuming that the internal master logic is clocked from Clk, it is acceptable for the master's outputs to be dependent on both the current state, the internal RdReq and WrReq signals, and the slave's outputs, since all of these are synchronous to Clk. Figure 37 shows a Mealy state machine for the OCP master. The assumptions about the internal master logic are the same as in "Sequential Master" on page 132, except that there is an additional acknowledge (Ack) signal output from the state machine to the internal master logic to indicate the completion of a transfer.

This state machine asserts MCmd in the same cycle that the request arrives from the internal master logic, so transfer latency is improved. In addition, the state machine is simpler than the Moore machine, requiring only two states instead of four. The request state is responsible for beginning and waiting for the end of the request phase. The wait resp state is only used on Read commands where the slave does not assert SResp in the same cycle it asserts SCmdAccept. The arcs described by dashed lines are optional features that allow a transition directly from the end of the response phase into the beginning of the request phase, which can reduce the turn-around delay on multi-cycle Read commands.

Figure 37 Combinational OCP Master



The cost of this approach is in timing. Since the master request phase outputs become valid a combinational logic delay after RdReq and WrReq, there is less setup time available to the slave. Furthermore, if the slave is capable of asserting SCmdAccept on the first cycle of the request phase, then the total path is:

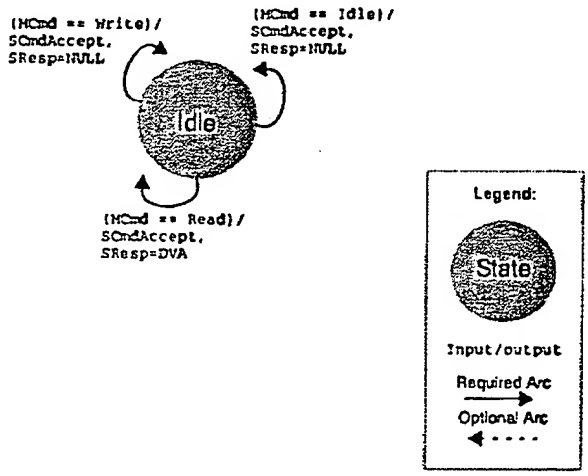
Clk -> (RdReq | WrReq) -> MCmd -> SCmdAccept -> Clk.

To successfully implement this path at high frequency requires careful analysis. The effort is appropriate for highly latency-sensitive masters such as CPU cores. At much lower frequencies, where area is often at a premium, the Mealy OCP master is attractive because it has fewer states and the timing constraints are much simpler to meet. This style of master design is appropriate for both the highest-performance and lowest-performance ends of the spectrum. A Moore state machine implementation may be more appropriate at medium performance.

Combinational Slave

Achieving peak OCP data throughput of one transfer per cycle is most commonly implemented using a combinational Mealy state machine implementation. If a slave can satisfy the request phase in the cycle it begins and deliver read data in the same cycle, the Mealy state machine representation is degenerate - there is only one state in the machine. The state machine always asserts SCmdAccept in the first request phase cycle, and asserts SResp in the same cycle for Read commands (assuming no response on writes as in the write posting model).

Figure 38 Combinational OCP Slave



The implementation shown in Figure 38, offers the ideal throughput of one transfer per cycle. This approach typically works best for low-speed I/O devices with FIFOs, medium-frequency but low-latency asynchronous SRAM controllers, and fast register files. This is because the timing path looks like:

Clk -> (master logic) -> MCmd -> (access internal slave resource) -> SResp -> Clk

This path is simplest to make when:

- Clk frequency is low
- Internal slave access time is small
- SResp can be determined based only on MCmd assertion (and not other request phase fields nor internal slave conditions)

To satisfy the access time and operating frequency constraints of higher-performance slaves such as main memory controllers, the OCP supports transfer pipelining. From the state machine perspective, pipelining splits the slave state machine into two loosely-coupled machines: one that accepts requests, and one that produces responses. Such machines are particularly useful with the burst extensions to the OCP.

OCP Subsets

It is possible to define simple interfaces - OCP subsets that are frequently required in complex SOC designs. The subsets provide simple interfaces for HW blocks, typically with one-directional, non-addressed, or odd data size capabilities. Since most of the OCP signals can be individually enabled or disabled, a variety of subsets can be defined. For the command set, any OCP command needs to be explicitly declared as supported by the core with at least one command enabled in a subset.

Some sample interfaces are listed in Table 24. For each example the non-default parameter settings are provided. The list of the corresponding OCP signals is provided for reference. Subset variants can further be derived from these examples by enabling various OCP extensions.

Table 24 OCP Subsets

Usage	Purpose	Non-default parameters	Signals
Handshake-only OCP	Simple request/acknowledge handshake, that can be used to synchronize two processing modules. Using OCP handshake signals with well-defined timing and semantics allows routing this synchronization process through an interconnect. The OCP command WR is used for requests, other commands are disabled.	read_enable=0, addr=0, mdata=0, sdata=0, resp=0	Clk, MCmd, SCmdAccept
Write-only OCP	Interface for cores that only need to support writes.	read_enable=0, sdata=0, resp=0	Clk, MAddr, MCmd, MData, SCmdAccept
Read-only OCP	Interface for cores that only need to support reads.	write_enable=0, mdata=0	Clk, MAddr, MCmd, SCmdAccept, SData, SResp
FIFO Write-only OCP	Interface to FIFO input.	read_enable=0 addr=0, sdata=0, resp=0	Clk, MCmd, MData, SCmdAccept
FIFO Read-only OCP	Interface to FIFO output.	write_enable=0, addr=0, mdata=0	Clk, MCmd, SCmdAccept, SData, SResp
FIFO OCP	Read and write interface to FIFO.	addr=0	Clk, MCmd, MData, SCmdAccept, SData, SResp

Simple OCP Extensions

The simple extensions to the OCP signals add support for higher-performance master and slave devices. Extensions include byte enable capability, multiple address spaces, and the addition of in-band socket-specific information to any of the three OCP phases (request, response, and datahandshake).

Byte Enables

Byte enable signals can be driven during the request phase for read or write operations, providing byte addressing capability, or partial OCP word transfer. This capability is enabled by setting the byteen parameter to 1.

Even for simpler OCP cores, it is good practice to implement the byte enable extension, making byte addressing available at the chip level with no restrictions for the host processors.

When a datahandshake phase is used (typically for a single request-multiple data burst), bursts must have the same byte enable pattern on all write data words. It is often necessary to send or receive write byte enables with the write data. To provide full byte addressing capability, the MDataByteEn field can be added to the datahandshake phase. This field indicates which bytes within the OCP data write word are part of the current write transfer.

For example, on its master OCP port, a 2D-graphics accelerator can use variable byte enable patterns to achieve good transparent block transfer performance. Any pixel during the memory copy operation that matches the color key value is discarded in the write by de-asserting the corresponding byte enable in the OCP word. Another example is a DRAM controller that, when connected to a x16-DDR device, needs to use the memory data mask lines to perform byte or 16-bit writes. The data mask lines are directly derived from the byte enable pattern.

Unpacking operations inside an interconnect can generate variable byte enable patterns across a burst on the narrower OCP side, even if the pattern is constant on the wider OCP side. Such unpacking operations may also result in a byte enable pattern of all zeros. Therefore, it is mandatory that slave cores fully support 0 as a legal pattern.

An OCP interface can be configured to include partial word transfers by using either the MByteEn field, or the MDataByteEn field, or both.

- If only MByteEn is present, the partial word is specified by this field for both read and write type transfers as part of the request phase. This is the most common case.
- If only MDataByteEn is present, the partial word is specified by this field for write type transfers as part of the datahandshake phase, and partial word reads are not supported.
- If both MByteEn and MDataByteEn are present, MByteEn specifies partial words for read transfers as part of the request phase, and is don't care for write type transfers. MDataByteEn specifies partial words for write transfers as part of the datahandshake phase, and is don't care for read type transfers.

Multiple Address Spaces

Logically separate memory regions with unique properties or behavior are often scattered in the system address map. The MAddrSpace signal permits explicit selection of these separate address spaces.

Address spaces typically differentiate a memory space within a core from the configuration register space within that same core, or differentiate several cores into an OCP subsystem including multiple OCP cores that can be mapped at non-contiguous addresses, from the top level system perspective.

Another example of the usage of the addrspace extension is the case of an OCP-to-PCI bridge, since PCI natively supports address spaces for configuration registers, memory spaces and I/o spaces.

In-Band Information

OCP can be extended to communicate information that is not assigned semantics by the OCP protocol. This is true for out-of-band information (flag, control/status signals) and also for in-band information. The designer or the chip level architect can define in-band extensions for the OCP phases.

The fields provided for that purpose are MReqInfo for the request phase, SRespInfo for the response phase, MDataInfo for the request phase or the datahandshake phase, and SDataInfo for the response phase. The presence and width of these fields can be controlled individually.

MReqInfo

Uses for MReqInfo can include user/supervisor storage attributes, cacheable storage attributes, data versus program access, emulation versus application access or any other access-related information, such as dynamic endianness qualification or access permission information.

MReqInfo bits have no assigned meanings but have behavior restrictions. MReqInfo is part of the request phase, so when MCmd is Idle, MReqInfo is a "don't care". When MCmd is asserted, MReqInfo must be held steady for the entire request phase. MReqInfo must be constant across an entire transaction, so the value may not change during a burst. This facilitates simple packing and unpacking of data at mismatched master/slave data widths, eliminating the transformation of information.

SRespInfo

Uses for SRespInfo can include error or status information, such as FIFO full or empty indications, or data response endianness information.

SRespInfo bits have no assigned meaning, but have behavior restrictions. SRespInfo is part of the response phase, so when SResp is NULL, SRespInfo is a "don't care". When SResp is asserted, SRespInfo must be held steady for the entire response phase. SRespInfo must be constant across an entire transaction, so the value may not change during responses to a burst. This facilitates simple packing and unpacking of data at mismatched master/slave data width, eliminating the transformation of information.

MDataInfo and SDataInfo

MDataInfo and SDataInfo have slightly different semantics. While they have no OCP-defined meaning, they may have packing/unpacking implications. MDataInfo and SDataInfo are only valid when their associated phase is asserted (request or datahandshake phase for MDataInfo, response phase for SDataInfo).

Uses for the MDataInfo and SDataInfo fields might include byte data parity in the low-order bits and/or data ECC values in the high-order (non-packable) bits.

The low-order `mdatainfobyte_wdth` bits of MDataInfo are associated with MData[7:0], and so forth for each higher-numbered byte within MData, so that the low-order `mdatainfobyte_wdth`*(data_wdth/8) bits of MDataInfo are associated with individual data bytes. Any remaining (upper) bits of MDataInfo cannot be packed or unpacked without further specification, although such bits may be used in cases with matched data width, where no transformation is required.

The difference between MReqInfo and the upper bits of MDataInfo is that only MDataInfo is allowed to change during a burst transfer. A similar relationship exists between SRespInfo and SDataInfo.

A slave should be operable when all bits of MReqInfo and MDataInfo are negated; in other words, any MReqInfo or MDataInfo signals defined by an OCP slave, but not present in the master will normally be negated (driven to logic 0) in the tie off rules. A master should be operable when all bits of SRespInfo and SDataInfo are negated.

Burst Extensions

A burst is basically a set of related OCP words. Burst framing signals provide a method for linking together otherwise-independent OCP transfers. This mechanism allows various parts of a system to optimize transfer performance using such techniques as SDRAM page-mode operation, burst transfers, and pre-fetching.

Burst support is a key enabler of SOC performance. The burst extension is frequently used in conjunction with pipelined master and slave devices. For a pipelined OCP device, the request phase is *de-coupled* from the response phase - that is, the request phase may begin and end several cycles before the associated response phase begins and ends. As such, it is useful to think of separate, loosely-coupled state machines to support either the master or the slave. Decoupling for pipeline efficiency remains true even if the OCP includes a separate datahandshake phase.

OCP-IP 2.0 Burst Capabilities

The OCP burst model includes a variety of options permitting close matching of core design requirements.

Exact Burst Lengths and Imprecise Burst Lengths

A burst can be either precise, of known length when issued by the initiator, or imprecise, the burst length is not specified at the start of the burst.

For precise bursts, MBurstLength is driven to the same value throughout the burst, but is really meaningful only during the first request phase. Precise bursts with a length that is a power-of-two can make use of the WRAP and XOR address sequence types.

For imprecise bursts, MBurstLength can assume different values for words in the burst, reflecting a best guess of the burst length. MBurstLength is a hint. Imprecise bursts are completed by a request with an MBurstLength=1, and cannot make use of the WRAP and XOR address sequence types.

Use the precise burst model whenever possible:

- It is compatible with the single request-multiple data model that provides advantages to the SOC in terms of performance and power.
- Since it is deterministic, it simplifies burst conversion. Restricting burst lengths to power-of-two values and using aligned incrementing bursts (by employing the burst_aligned parameter) also reduces the interconnect complexity needed to maintain interoperability between cores.

Address Sequences

Using the MBurstSeq field, the OCP burst model supports commonly-used address sequences. Benefits include:

- A simple incrementing scheme for regular memory type accesses
- A constant addressing mode for FIFO oriented targets (typically peripherals)
- Wrapping on power-of-two boundaries
- XOR for processor cache line fill

User-defined sequences can also be defined. They must be carefully documented in the core specification, particularly the rules to be applied when packing or unpacking. The address behavior for the different sequence types is:

INCR
: Each address is incremented by the OCP word size. Used for regular memory type accesses, SDRAM, SRAM, and burst Flash.

STRM
: The address is constant during the burst. Used for streaming data to or from a target, typically a peripheral device including a FIFO interface that is mapped at a constant address within the system.

DFLT1
: User-specified address sequence. Maximum packing is required.

DFLT2
: User-specified address sequence. Packing is not allowed.

WRAP

Similar to INCR, except that the address wraps at aligned MBurstLength * OCP word size. This address sequence is typically used for processor cache line fill. Burst length is necessarily a power-of-two, and the burst is aligned on its size.

XOR

Addr = BurstBaseAddress +(index of first request in burst) ^ (current word number). XOR is used by some processors for critical-word first cache line fill from wide and slow memory systems.

UNKN

Indicates that there is no specified relationship between the addresses of different words in the burst. Used to group requests within a burst container, when the address sequence does not match the pre-defined sequences. For example, an initiator can group requests to non-consecutive addresses on the same SDRAM page, so that the target memory bandwidth can be increased.

For targets that have support for some burst sequence, adding support for the UNKN burst sequence can improve the chances of interoperability with other cores and can ease verification since it removes all requirements from the address sequence within a burst.

For single requests, MBurstSeq is allowed to be any value that is legal for that particular OCP interface configuration.

The INCR, DFLT1, WRAP, and XOR burst sequences are always considered packing, whereas STRM and DFLT2 sequences are non-packing. Transfers in a packing burst sequence are aggregated / split when translating between OCP interfaces of different widths while transfers in a non-packing sequence are filled / truncated.

The packing behavior of a bridge or interconnect for an UNKN burst sequence is system-dependent. A common policy is to treat an UNKN sequence as packing in a wide-to-narrow OCP request width converter, and as non-packing in a narrow-to-wide OCP request width converter.

Single Request, Multiple Data Bursts for Reads and Writes

A burst model of this type can reduce power consumption, bandwidth congestion on the request path, and buffering requirement at various locations in the system. This model is only applicable for precise bursts, and assumes that the target core can reconstruct the full address sequence using the code provided in the MBurstSeq field.

While the model assumes that the datahandshake extension is on, for those cores that do not accept the first-data word with the corresponding request, datahandshake can increase design and verification complexity.

For such cores, use the OCP parameter `reqdata_together` to specify the fixed timing relationship between the request and datahandshake phases. When `reqdata_together` is set, the request and datahandshake phases of the first transfer in the single request / multiple data write-type burst begin and end together.

Unit of Atomicity

Use this option when there is a requirement to limit burst interleaving between several threads. Specifying the atomicity allows the master to define a transfer unit that is guaranteed to be handled as an atomic group of requests within the burst, regardless of the total burst length. The master indicates the size of the atomic unit using the MAtomicLength field.

Burst Framing with all Transfer Phases

Without burst framing information, cores and interconnects incorporate counters in their control logic: To limit this extra gate count and complexity, enable end-of-burst information for each phase. Use MReqLast to specify the last request within a burst, SRespLast to specify the last response within a burst, and MDataLast to specify the last write data during the datahandshake phase.

Compatibility with the OCP 1.0 Burst Model

The OCP-IP 2.0 burst model replaces the OCP-IP 1.0 model, providing a superset in terms of available functionality. To maintain interoperability between cores using the OCP-IP 1.0 burst and cores using the OCP-IP 2.0 bursts requires a thin adaptation layer. Guidelines for the wrapping logic are described in this section.

1.0 Master to 2.0 Slave

For converting an OCP 1.0 burst into an OCP 2.0 burst the suggested mapping is:

- MBurstPrecise is available only when the OCP 1.0 `burst_aligned` parameter is set. When set, all incrementing bursts once converted to OCP 2.0 stay precise. Any other OCP 1.0 burst type is mapped to an imprecise burst. When `burst_aligned` is not set, MBurstPrecise is tied off to 0, so all burst are imprecise.

- MBurstSeq is derived from MBurst as follows:

  MBurstSeq = INCR for MBurst {CONT, TWO, FOUR, EIGHT}
  STRM for MBurst {STRM}
  DFLT1 for MBurst {DFLT1}
  DFLT2 for MBurst {DFLT2}

  The logic must guarantee that MBurstSeq is constant during the whole burst and must continue driving that MBurstSeq when MBurst=LAST is detected.

- The value of MBurstLength is derived as follows:

  MBurstLength = 8 for MBurst {EIGHT}
  4 for MBurst {FOUR}
  2 for MBurst {TWO, CONT, DFLT1, DFLT2, STRM}
  1 for MBurst {LAST}

For precise bursts, MBurstLength is held constant for the entire burst. For imprecise bursts, a new MBurstLength can be derived for each transfer.

- MReqLast is derived from MBurst - it is set when MBurst is LAST.
- SRespLast has no equivalent in OCP1.0, and is discarded by the wrapping logic.
- If required, MDataLast must be generated from a counter or from a queue updated during the request phase.

2.0 Master to 1.0 Slave

For converting an OCP 2.0 burst into an OCP 1.0 burst the suggested mapping is:

- MBurst is derived from MBurstPrecise, MBurstSeq and MBurstLength, as follows:

```
MBurst =
If MBurstPrecise
    if MBurstSeq {INCR}
        EIGHT if MBurstLength >= 8 at start of burst
        FOUR if MBurstLength >= 4 at start of burst
        TWO if MBurstLength >= 2 at start of burst
        load counter with MBurstLength at start of burst,
        decrement counter after every transfer
        subsequent MBurst are generated from counter logic
        LAST when counter==1
    else if MBurstSeq {DFLT1, DFLT2, STRM}
        same as MBurstSeq, except when counter==1, must be LAST
    else if MBurstSeq {WRAP, XOR, UNKN}
        LAST always: map to consecutive non-burst single transactions

Else if not MBurstPrecise
    if MBurstSeq {INCR}
        EIGHT if MBurstLength >= 8
        FOUR if MBurstLength >= 4
        TWO if MBurstLength >= 2
        LAST if MBurstLength == 1
    else if MBurstSeq {DFLT1, DFLT2, STRM}
        LAST if MBurstLength == 1
        same as MBurstSeq if MBurstLength != 1
    else if MBurstSeq {WRAP, XOR, UNKN}
        LAST always (map to non-burst)
```

- MAtomicLength, MReqLast, and MDataLast have no equivalents in OCP 1.0, and are discarded by the wrapping logic.
- SRespLast must be generated from counter logic.

The logic described above is not suitable if the OCP 2.0 master generates single request / multiple data bursts. In that case, more complex conversion logic is required.

Threads and Connections

Thread extensions add support for concurrency. Without these extensions, the OCP protocol enforces strict ordering constraints; each transfer request must be completed by the slave in the same order presented by the master. This restriction ensures proper system functionality (that is, an OCP read that occurs after a write to the same address will return the new data) and simplifies the transfer tracking requirements of implementing pipelining. Threading extensions enhance the basic transfer capability by introducing the concept of threads.

Threads

The thread capability relies on a thread ID to identify and separate independent transfer streams (threads). The master labels each request with the thread ID that it has assigned to the thread. The thread ID is passed to the slave on MThreadID together with the request (MCmd). When the slave returns a response, it also provides the thread ID (on SThreadID) so the master knows which request is now complete.

The transfers in each thread must remain in-order with respect to each other (as in the basic OCP), but the order between threads can change between request and response.

The thread capability allows a slave device to optimize its operations. For instance, a multi-bank SDRAM controller could respond to a second read request referencing an open page before opening a new page in a different bank to service a first read on a different thread.

As routing congestion and physical effects become increasingly difficult at the back-end stage of the ASIC process, multithreading offers a powerful method of reducing wires. Many functional connections between initiator and target pairs do not require the full bandwidth of an OCP link, so sharing the same wires between several connections, based on functional requirements and floor planning data, is an attractive mechanism to perform gate count versus performance versus wire density trade-offs.

Multi-threaded behavior is most frequently implemented using one state machine per thread. The only added complexity is the arbitration scheme between threads. This is unavoidable, since the entire purpose for building a multi-threaded OCP is to support concurrency, which directly implies contention for any shared resources.

The MDataThreadID signal simplifies the implementation of the datahandshake extension along with threading, by providing the thread ID associated with the current write data transfer. When datahandshake is enabled, but sdatathreadbusy is disabled, the ordering of the datahandshake phases must exactly match the ordering of the request phases.

The thread busy signals provide status information that allows the master to determine which threads will not accept requests. That information also allows the slave to determine which threads will not accept responses. These signals provide for cooperation between the master and the slave to ensure that requests are not presented on busy threads.

While multithreading support has a cost in terms of gate count (buffers are required on a thread-per-thread basis for maximum efficiency), the protocol can ensure that the multi-threaded interface is non-blocking.

Blocked OCP interfaces introduce a thread dependency. If thread X cannot proceed because the OCP interface is blocked by another thread, Y that is dependent on something downstream that cannot make progress until thread X makes progress, there is a classic circular wait condition that can lead to deadlock.

In the OCP-IP1.0 specification, the semantics of SThreadBusy and MThread-Busy allow these signals to be treated as hints. To guarantee that a multi-threaded interface does not block, both master and slave need to be held to tighter semantics.

OCP-IP 2.0 allows cores to specify that they are obeying exact thread busy semantics. This process enables tighter protocol checking at the interface and guarantees that a multi-threaded OCP interface is non-blocking. Parameters to enable these extensions are `sthreadbusy_exact`, `sdatathreadbusy_exact`, and `mthreadbusy_exact`. There is one parameter for each of the OCP phases, request, datahandshake (assuming separate datahandshake) and response (assuming response flow control). The following conditions are true:

- On an OCP interface that satisfies `sthreadbusy_exact` semantics, the master is not allowed to issue a command to a busy thread.
- On an OCP interface that complies with `sdatathreadbusy_exact` semantics, the master is not allowed to issue write data to a busy thread.
- On an OCP interface that complies with `mthreadbusy_exact` semantics, the slave is not allowed to issue a response to a busy thread.

These rules allow the phase accept signals (SCmdAccept, SDataAccept or MRespAccept) to be tied off to 1 on multi-threaded interfaces for which the corresponding phase handshake satisfies exact thread busy semantics. By eliminating an additional combinational dependency between master and slave, an exact thread busy based handshake can be considered as a substitute for the standard request/accept protocol handshake. For more information, see "Multi-Threaded OCP Implementation" on page 149.

Intra-Phase Signal Relationships on a Multithreaded OCP

This section extends the timing discussion of the "Basic OCP" section, to a multithreaded interface. The ordering and timing relationships between the signals within an OCP phase are designed to be flexible. As described in "Request Phase", it is legal for SCmdAccept to be driven either combinationally, dependent upon the current cycle's MCmd or independently from MCmd, based on the characteristics of the OCP slave. Some restrictions are required to ensure that independently-created OCP masters and slaves will work together. For instance, the MCmd cannot respond to the current state of SCmdAccept; otherwise, a combinational cycle could occur.

Request Phase

If enabled, a slave's SThreadBusy request phase output should not depend upon the current state of any other OCP signal. SThreadBusy should be stable early enough in the cycle so that the master can factor the current SThreadBusy into the decision of which thread to present a request; that is, all of the master's request phase outputs may depend upon the current SThreadBusy. SThreadBusy is a hint so the master is not required to include a combinational path from SThreadBusy into MCmd, but such paths become unavoidable if the exact semantics apply (`sthreadbusy_exact = 1`). In that case the slave must guarantee that SThreadBusy becomes stable early in the OCP cycle to achieve good frequency performance. A common goal is that SThreadBusy be driven directly from a flip-flop in the slave.

A master's request phase outputs should not depend upon any current slave output other then SThreadBusy. This ensures that there is no combinational loop in the case where the slave's SCmdAccept depends upon the current MCmd.

If a slave's SCmdAccept request phase output is based upon the master's request phase outputs from the current cycle, there is a combinational path from MCmd to SCmdAccept. Otherwise, SCmdAccept may be driven directly from a flip-flop, or based upon some other OCP signals. It is legal for SCmdAccept to be derived from MRespAccept. This case arises when the slave delays SCmdAccept to force the master to hold the request fields for a multi-cycle access. Once read data is available, the slave attempts to return it by asserting SResp. If the OCP has MRespAccept enabled, the slave then must wait for MRespAccept before negating SResp, so it may need to continue to hold off SCmdAccept until it sees MRespAccept asserted.

While the phase relationships of the OCP specification do not allow the response phase to end before the request phase, it is legal for both phases to complete in the same OCP cycle.

The worst-case combinational path for the request phase could be:

```
Clk -> SThreadBusy -> MCmd -> SResp -> MRespAccept -> SCmdAccept -> Clk
```

The preceding path has too much latency at typical clock frequencies, so must be avoided. Fortunately, a multi-threaded slave (with SThreadBusy enabled) is not likely to exhibit non-pipelined read behavior, so this path is unlikely to prove useful. Slave designers need to limit the combinational paths visible at the OCP. By pipelining the read request, the previous path could be:

```
Clk -> SThreadBusy -> MCmd -> Clk
Clk -> SCmdAccept -> Clk      # Slave accepts if pipeline reg empty
Clk -> SResp -> Clk
Clk -> MRespAccept -> Clk     # Master accepts independent of SResp
```

Response Phase

If enabled, a master's MThreadBusy response phase output should not be dependent upon the current state of any other OCP signal. From the perspective of the OCP, MThreadBusy should become stable early enough in the cycle

that the slave can factor the current MThreadBusy into the decision on which thread to present a response; that is, all of the slave's response phase outputs may depend upon the current MThreadBusy. If MThreadBusy is simply a hint (in other words mthreadbusy_exact = 0) the slave is not required to include a combinational path from MThreadBusy into SResp, but such paths become unavoidable if the exact semantics apply (mthreadbusy_exact = 1). In that case the master must guarantee that MThreadBusy becomes stable early in the OCP cycle to achieve good frequency performance. A common goal is that MThreadBusy be driven directly from a flip-flop in the master.

The slave's response phase outputs should not depend upon any current master output other than MThreadBusy. This ensures that there is no combinational loop in the case where the master's MRespAccept depends upon the current SResp.

The master's MRespAccept response phase output may be based upon the slave's response phase outputs from the current cycle or not. If this is true, there is a combinational path from SResp to MRespAccept. Otherwise, MRespAccept can be driven directly from a flip-flop; MRespAccept should not be dependent upon other master outputs.

Datahandshake Phase

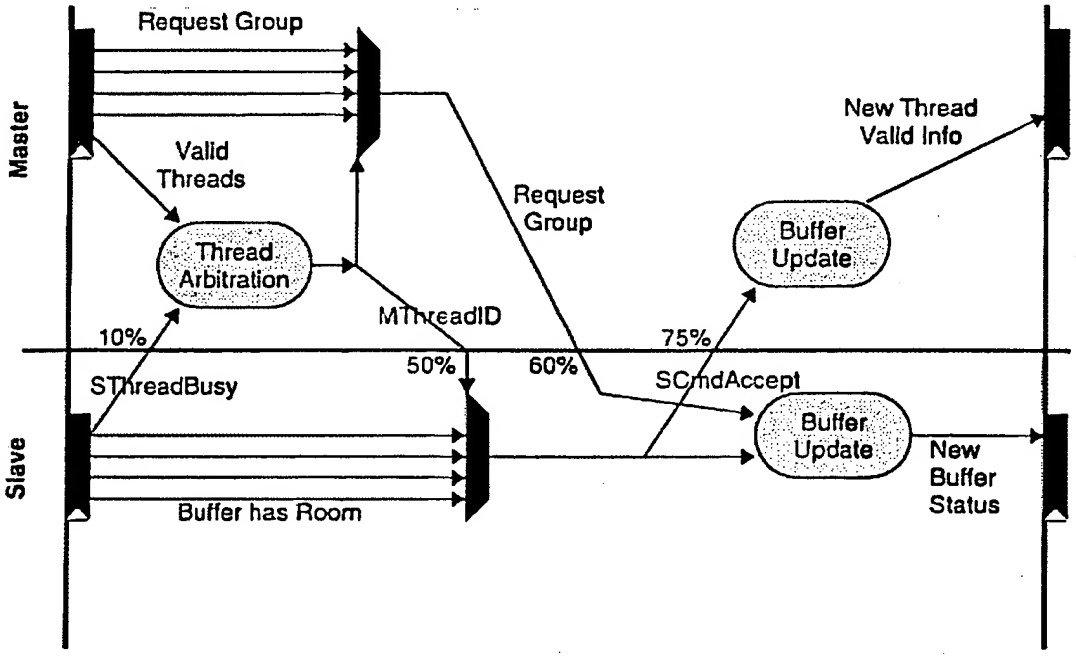
If enabled, a slave's SDataThreadBusy datahandshake phase output should not depend upon the current state of any other OCP signal. SDataThreadBusy should be stable early enough in the cycle so that the master can factor the current SDataThreadBusy into the decision of which thread to present a data; that is, all of the master's data phase outputs may depend upon the current SDataThreadBusy. If SDataThreadBusy is simply a hint (in other words sdatathreadbusy_exact = 0) the master is not required to include a combinational path from SDataThreadBusy into MDataValid, but such path becomes unavoidable if the exact semantics apply (sdatathreadbusy_exact = 1). In that case, the slave must guarantee that SDataThreadBusy becomes stable early in the OCP cycle to achieve good frequency performance. A common goal is that SDataThreadBusy be driven directly from a flip-flop in the slave.

The master's datahandshake phase outputs should not depend upon any current slave output other than SThreadBusy. This ensures that there is no combinational loop in the case where the slave's SDataAccept depends upon the current MDataValid. The slave's SDataAccept output may or may not be based upon the master's datahandshake phase outputs from the current cycle. In the former case, there is a combinational path from MDataValid to SDataAccept. In the latter case, SDataAccept should be driven directly from a flip-flop; SDataAccept should not be dependent upon other master outputs.

Multi-Threaded OCP Implementation

Figure 39 on page 150 shows the typical implementation of the combinational paths required to make a multi-threaded OCP work within the framework set by Level-2 timing. While the figure shows a request phase, similar logic can be used for the response and datahandshake phases. The top half of the figure shows logic in the master; the bottom half shows logic in the slave. The width of the figure represents a single OCP cycle.

Figure 39 Multithreaded OCP Interface Implementation



Slave

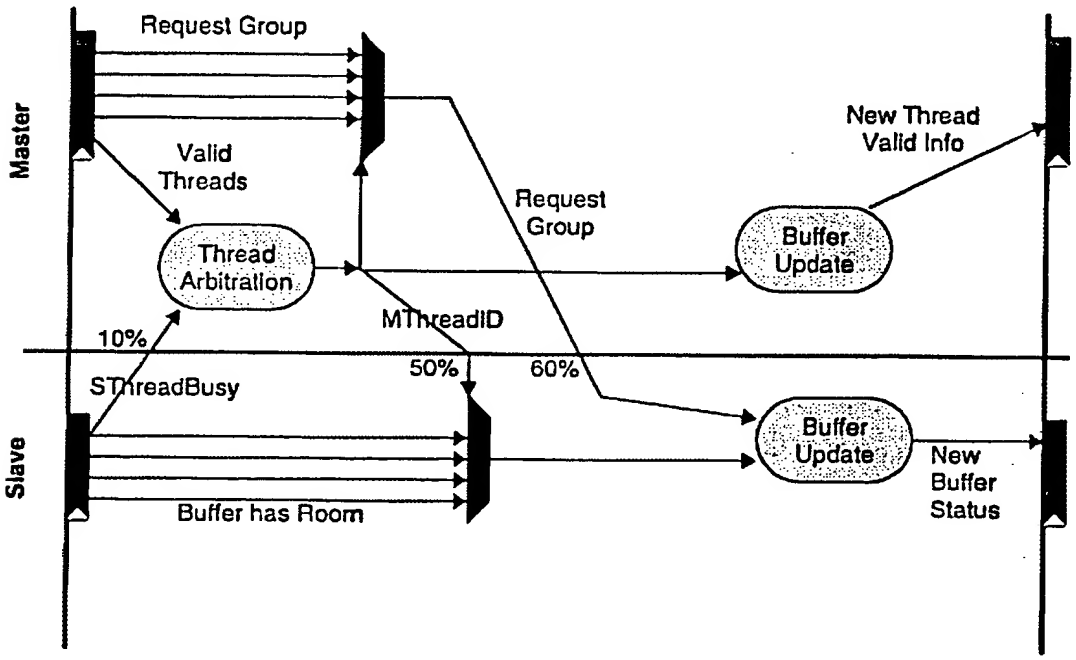
Information about space available on the per-port buffers comes out of a latch and is used to generate SThreadBusy information, which must be generated within the initial 10% of the OCP cycle (as described in "Level2 Timing" on page 162). These signals are also used to generate SCmdAccept: if a particular port has room, a command on the corresponding thread is accepted. The correct port information is selected through a multiplexer driven by MThreadID at 50% of the clock cycle, making it easy to produce SCmdAccept by 75% of the OCP cycle. When the request group arrives at 60% of the OCP cycle, it is used to update the buffer status, which in turn becomes the SThreadBusy information for the next cycle.

Master

The master keeps information on what threads have commands ready to be presented (thread valid bits). When SThreadBusy arrives at 10% of the OCP clock, it is used to mask off requests, that is any thread that has its SThreadBusy signal set is not allowed to participate in arbitration for the OCP. The remaining thread valid bits are fed to thread arbitration, the result of which is the winning thread identifier, MThreadId. This is passed to the slave at 50% of the OCP clock period. It is also used to select the winning thread's request group, which is then passed to the slave at 60% of the clock period. When the SCmdAccept signal arrives from the slave, it is used to compute the new thread valid bits for the next cycle.

The request phase in Figure 39 assumes a non-exact thread busy model. The exact model shown in Figure 40 is similar, but SCmdAccept is tied off to 1, so any request issued to a non-busy thread is accepted in the same cycle by the slave.

Figure 40 Multithreaded OCP Interface with threadbusy_exact



Connections

In multi-threaded, multi-initiator systems, it is frequently useful to associate a transfer request with a thread operating on a particular initiator. Initiator identification can enable a system to restrict access to shared resources, or foster an error logging mechanism to identify an initiator whose request has created an error in the system. For devices where these concerns are important, the OCP extensions support connections.

Connections are closely related to threads, but can have end-to-end meaning in the system, rather than the local meaning (that is, master to slave) of a thread.

The connection ID and thread ID seem to provide similar functionality, so it is useful to consider why the OCP needs both. A *thread ID* is an identifier of local scope that simply identifies transfers between the master and slave. In contrast, the *connection ID* is an identifier of global scope that identifies transfers between a system initiator and a system target. A thread ID must be small enough (that is, a few bits) to efficiently index tables or state machines within the master and slave. There are usually more connection IDs in the system than any one slave is prepared to simultaneously accept. Using a connection ID in place of a thread ID requires expensive matching logic in the master to associate the returned connection ID (from the slave) with specific requests or buffer entries.

Using a networking analogy, the thread ID is a level-2 (data link layer) concept, whereas the connection ID is more like a level-3 (transport/session layer) concept. Some OCP slaves only operate at level-2, so it doesn't make sense to burden them or their masters with the expense of dealing with level-3 resources. Alternatively, some slaves need the features of level-3 connections, so in this case it makes sense to pass the connection ID through to them.

A connection ID is not usually provided by an initiator core on its OCP interface but is allocated to that particular initiator in the interconnect logic of the system. The connection ID is system-specific, not core-specific; only the system integrator has the global knowledge of the number of initiators instantiated in the application, and what the requirements are in terms of differentiation.

As an exception to that rule, if the global interconnect consists of multiple hierarchical structures, a complete subsystem can be integrated (including another interconnect with multiple embedded initiators). In that case, the OCP interface between the two interconnects should implement the connid extension, so that the end-to-end meaning of that OCP field can be preserved at the system level.

For a target core, the connid extension is included when such features as access control, error logging or similar initiator-related features require initiator identification.

OCP Specific Features

Write Semantics

The OCP-IP 1.0 Specification includes a Write command without a response. From the initiator standpoint, it completes once the request has been accepted, so follows a posted write model. The 2.0 release semantics specify whether a write is posted or not. A non-posted write model is preferred whenever the originator of the request must be aware of the completion of its write command; An example is clearing an interrupt in a peripheral module using a write command. There the processor must be sure that the interrupt line has been effectively released before it can acknowledge the interrupt service in the chip-level interrupt controller.

The 2.0 extension separates the concept of the posting semantics from the concept of responses on writes in the following ways:

- A write with a response could have posted semantics in a system (so that a response is returned immediately) or it could have non-posted semantics (so that a response is returned only after the write is completed at the final target).
- A write without a response normally has posted semantics and carries forward the OCP-IP 1.0 specification for backward compatibility.

- A write without a response can be assigned non-posted semantics by not accepting the command until the write has completed, but this is not recommended since it de-pipelines the OCP interface. Since posting makes sense at a system level, adopting a delayed-SCmdAccept scheme can only be efficient locally, with no guarantee of the non-posting semantics at the system level.

The `writeresp_enable` parameter controls whether writes have responses or not. `writenonpost_enable` controls whether the interface supports the WriteNonPost command, giving the initiator core the ability to launch two different types of writes. Table 25 summarizes the choices.

Table 25 Write Parameters

	writenonpost_enable	
writeresp_enable	**0**	**1**
0	Simple posted write model, corresponds to OCP-IP1.0 spec	Illegal
1	Initiator core has one flavor of writes (WR), system integrator decides where to post the write requests	Initiator core has two flavors of writes (WR and WRNP), system integrator decides where to post the two different write requests

By separating whether writes have responses (`writeresp_enable`) from whether the core has control over where the responses are generated (`writenonpost_enable`), the OCP specification provides the following features:

- The simple, posted model remains intact. The simplest cores only implement WR, and need not worry about write responses.
- Cores that can generate or use write responses should enable write responses, providing support for in-band error reporting on write commands. The read and write state machines are duplicated from the standpoint of flow control, producing a simpler design. Such cores would normally only implement the WR command. In this case, the system integrator is in control of where in the write path the write response is generated, allowing a choice of the level of posting based upon performance and coherence trade-offs.
- Cores that can distinguish between performance and coherence (really only CPUs and bridges) can enable WRNP to implement dynamic choice between WR and WRNP. The additional signaling gives the system integrator the dynamic information needed to choose the posting point as the CPU requests. The only practical difference between WR and WRNP at the protocol level is the expected latency between request and response. This permits some embedded CPUs to achieve high performance – particularly as interconnects become pipelined and posting buffers are needed.

When the writeresp_enable parameter is enabled, responses are required for any write command issued on the OCP, including WR, WRNP, but also BCST and WRC.

Use of the Broadcast command must be limited to specific category of designs (some interconnect designs may benefit from simultaneous update through distributed registers). It is not expected that standard cores support the command.

Lazy Synchronization

Most processors support semaphores through a read-modify-write type of instruction and swap, test-and-set, etc. Using an OCP interconnect, these instructions are mapped onto a pair of OCP commands. A RDEX command sets a lock to the memory location, followed by a WR (or WRNP) command to release the lock. The system must ensure that no other thread will be granted access to that memory location between the RDEX and the unlocking WR.

Because the Write that clears the lock must immediately follow the ReadEx (on the same thread), only a limited number of operations can be performed by a processor between RDEX and WR. Because competing requests to the locked location are also blocked from proceeding until the lock is unset, you could lock part of the interconnect for the duration of the RDEX-WR or WRNP pair. Such a mechanism, often referred to as *locked synchronization*, is efficient for handling exclusive accesses to a shared resource, but can result in a significant performance loss when used extensively.

For these reasons, some processors use non-blocking instructions for semaphore handling, breaking the atomicity of the exclusive read/write pair. For the processor, this allows other instructions to be executed by the processor between the read and write accesses. For the system interconnect, it allows requests from other threads to be inserted between the read and write commands. Referred to as lazy synchronization, this mechanism requires new read and write semantics, commonly known as LL/SC semantics, for Load-Linked and Store-Conditional.

OCP-IP 2.0 support for lazy synchronization uses the ReadLinked, and WriteConditional commands. A single OCP parameter rdlwrc_enable is set to 1, to enable the commands. Because some processors might use both semantics (locked and lazy), the OCP interface supports RDEX, RDL, WR(WRNP) and WRC.

The system relies upon the existence of monitor logic, that can be located either in the system interconnect, or in the memory controller. The ReadLinked command sets a reservation tag in the logic, associating the accessing thread with a particular address. The WriteConditional command, being transmitted on the same thread, is locally transformed into a memory write access only if the reservation tag is still set when the command is received. As the tagged address is not locked, the tag can be reset by competing traffic directed to the same location from other threads between RDL and WRC.

Consequently, the WRC command expects a response from the monitor logic, reflecting whether the write operation has been performed. To answer that requirement, OCP-IP 2.0 provides the value, FAIL for the SResp field (meaning that `writeresp_enable` is on if `rdlwrc_enable` is on). WRC is the only OCP-IP 2.0 command that makes use of the FAIL code, though new commands in future revisions may. FAIL responses are frequently received in a system using lazy synchronization that operates normally. Do not confuse FAIL with SResp=ERR, which effectively signals a system interconnect error or a target error.

Both RDL and WRC commands assume a single transaction model and cannot be used in a burst.

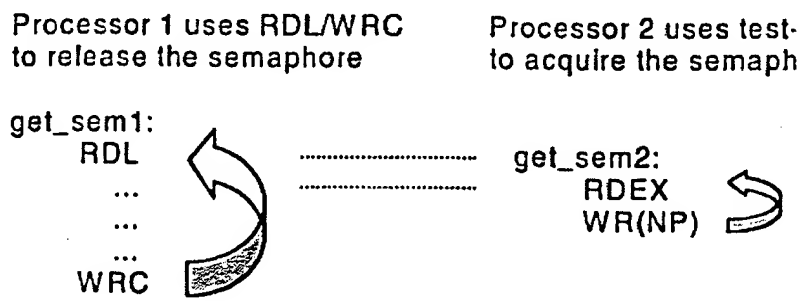
The semantics of lazy synchronization are defined in the specification section of this document (Part I). Some specific sequences resulting from the usage of the RDL and WRC semantics are:

- A thread can issue more than one RDL before issuing a WRC, or issue more than one RDL without issuing WRC. Whether the subsequent RDL clears the reservation tag or sets a new one is implementation-specific, depending on the number of hardware monitors. At least one monitor per thread is required.
- If a thread issues a WR or WRNP command to an address it previously tagged with a RDL command, the write access clears all reservation tags from other threads for the same address (but not its own reservation tag).
- If a thread issues a WRC without having issued a RDL, the WRC will fail.
- If a thread issues a RDEX between the RDL and WRC, the RDEX is executed, sets the lock and waits for the corresponding write to clear the lock. RDL-WRC reservation tags will not be affected by the RDEX. The WR or WRNP that clears the lock, also clears any reservation tag set by other initiators for the same address (with the same MAddr, MByteEn and MAddrSpace if applicable).

Because competing requests of any type from other threads to a locked (by RDEX) location are blocked from proceeding until the lock is unset, a RDEX command being issued between RDL and WRC commands, also blocks the WRC until the WR or WRNP command clearing the lock is issued. This favours RDEX-WR or WRNP sequence over RDL-WRC, in the sense that competing RDEX-WR or WRNP and RDL-WRC sequences will always result in having the RDEX-WR or WRNP sequence win.

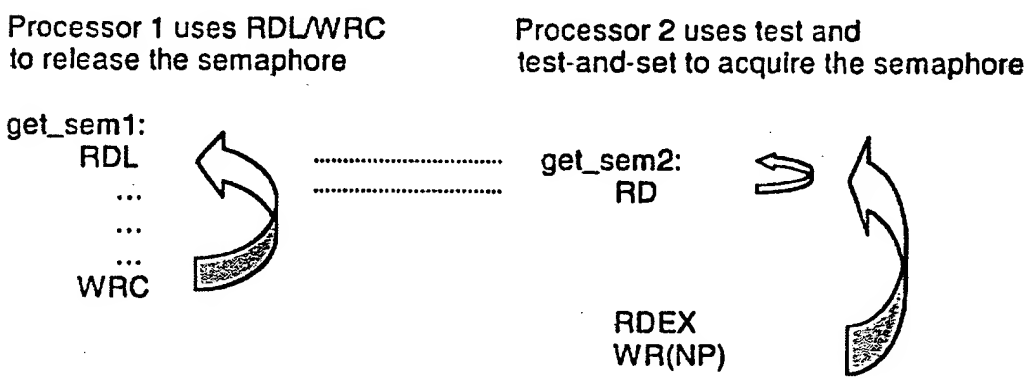
Incorrect use of the two synchronization mechanisms can result in deadlock. The sequence of commands shown in Figure 41 might result in a deadlock. In this example Processor 1 tries to release the semaphore using RDL-WRC commands, Processor 2 tries to acquire the semaphore using RDEX-WR or WRNP commands. The RDEX-WR or WRNP sequence always occurs between the RDL and WRC. Because the WR or WRNP clearing the lock in Processor2 will also clear the reservation tag for Processor 1, the RDL-WRC sequence will never succeed. Processor 1 will never be able to release the lock or Processor2 to acquire it.

Figure 41 Synchronization Deadlock



The deadlock depicted in Figure 41 is a result of bad programming in Processor 2, and is very unlikely to happen in a real application environment. As shown in Figure 42, to achieve forward progress, Processor 2 should read the semaphore value and wait for the semaphore to be free before trying to retrieve it by issuing a RDEX-WR or WRNP.

Figure 42 Correct Synchronization Sequence



OCP and Endianness

As described in "Endianness" on page 41, OCP is nearly endian-neutral. While OCP specifies a byte address on MAddr, the address must be aligned to the data width of the interface. Sub-word quantities are specified using one bit for each enabled byte in the transfer on MByteEn or MDataByteEn.

While the bit ordering of OCP fields is consistently described in a little-endian fashion, this is conventional, where even big-endian systems tend to number their bits little-endian. Similarly, the MByteEn numbering seems to imply a little-endian byte ordering, but is simply intended to maintain consistency. For example, MByteEn[m] refers to the byte transferred on MData/SData[(8m+7):8m] (provided m < data width/8), regardless of the effective transfer endianness attributes.

If the master OCP and the slave OCP are the same data width, endianness does not matter. Addresses, data, and byte enables must remain consistent across both interfaces. (There are exceptions, since packed sub-word data objects should be swapped if the endianness does not match. OCP does not carry the required signaling to determine sub-word sizes, so full-word transfers must be assumed.)

Endianness problems arise as soon as one looks to connect a master and slave with different data widths. The narrow side has extra (non-zero) address bits, since its word-aligned addresses do not force as many bits to be zero. The wide side has extra byte lanes to carry its wider words. The association of the extra address bits (narrow side) with the extra byte lanes (wide side) specifies an endianness.

To bridge interfaces that suffer from mismatched data widths, packing and unpacking is required. Data width conversion must make some assumptions about the correspondence between the MAddr least-significant bits and the MByteEn field.

If the association maps the low-order byte lanes to lower addresses, the data width conversion is performed in a little-endian manner. If the association maps the high-order byte lanes to lower addresses, the data width conversion is performed in a big-endian manner. This operation is absolutely not an endianness conversion, but rather an endianness-aware packing or unpacking operation, so that the transaction endianness is preserved across the data width converter.

There is no attempt to perform any endian conversion in hardware. Rather, the goal is to enable interconnects that are essentially endian-neutral, but become endian-adaptive to match the endianness of the attached entities. This implies that the native endianness of an OCP core must be specified. OCP-IP 2.0 captures that property using the `endian` parameter, which can take four values:

LITTLE
: Qualifies little-endian only cores

BIG
: Qualifies big-endian only cores

BOTH
: Qualifies cores that can change endianness:

  - Based upon an external input such as a CPU that statically selects its endianness at boot time
  - Based upon an internal configuration register such as a DMA engine, that generates OCP read and write requests in accordance with the endianness of the target, as stated by the DMA programmer
  - Cores that support dynamic endianness

NEUTRAL
Qualifies cores that have no inherent endianness. Examples are simple memory devices that only work with full OCP-word quantities, or peripheral devices, the endianness of which can be controlled by the software device driver.

While not supported by the OCP-IP 2.0 standard set of features, it is possible to define a dynamic endian-aware interconnect using in-band information. By specifying the parameters `reqinfo` (for request packing / unpacking control), `mdatainfo` (for data packing / unpacking control when `datahandshake` is enabled), and `respinfo` (for response packing / unpacking control), the definition of all these qualifiers is then platform-specific.

Sideband Signals

The sideband signals provide a means of transmitting control-oriented information. Since the signals are rarely performance sensitive, implementors are strongly encouraged to ensure that all sideband signals are driven stable very early in the OCP clock cycle; that is, that sideband outputs come directly out of core flip-flops. Sideband inputs should similarly be allowed to arrive very late in the OCP clock cycle; that is, sideband inputs should be registered almost immediately by the receiving core.

Cores that do not implement this conservative timing may require modification to achieve timing convergence.

Reset Handling

Many systems are fully satisfied with a single reset signal applied to both the master and the slave on an OCP interface. Either the master or the slave can drive the reset, or a third entity, such as a chip level reset manager, can provide it to both master and slave.

In some situations, it is more convenient for the master and slave to employ their own reset domains and communicate these internal resets to one another. The OCP interface is unable to communicate until both sides are out of reset since the side still in reset may be driving undetermined values (X) on their OCP outputs and cause problems for the side that is already out of reset. Examples of cases where this might arise are:

- A core with multiple OCP interfaces that are connected to different interconnects, which are each in different reset domains, plus the core has its own internal reset domain.
- Two connected interconnects with both acting as initiators of transfers and each with their own reset domain.

Adding a second reset signal to the interface allows each master and slave to have both a reset output and input. The composite reset state for the OCP interface is established as the combination of the two resets, so that either side (or both) asserting reset causes the interface to be in reset. While in reset, the existing rules about the interface state and signal values apply.

Either MReset_n or SReset_n must be present on any OCP interface. Compatibility between different reset configurations of master and slave interfaces is shown in Table 26.

Table 26 Reset Configurations

Master / Slave	sreset=1, mreset=0	sreset=0, mreset=1	sreset=1, mreset=1
sreset=0, mreset=1	Dual resets driven by the same 3rd party	Single reset driven by master	Single reset driven by master (SReset_n input tied off to 1)
sreset=1, mreset=0	Single reset driven by slave	Incompatible	Incompatible
sreset=1, mreset=1	Single reset driven by slave (MReset_n input tied off to 1)	Incompatible	Dual resets

The rules describing this table can be stated as follows.

- Either mreset or sreset or both must be set to 1 for each core.
- The default (and only) tie-off value for MReset_n and SReset_n is 1.
- If mreset is set to 1 for the master and mreset is set to 0 for the slave, the reset configurations are incompatible.
- If sreset is set to 1 for the slave and sreset is set to 0 for the master, the reset configurations are incompatible.

Cores with a reset input are always interoperable with any other core. Add a reset output if it is needed by the core or subsystem to assure proper operation. Typically this is because both sides need to know about the reset state of the other side, or because the overall system does not function properly if the core or subsystem is in reset, while the OCP interface is not in reset.

Compatibility with OCP-IP 1.0

OCP-IP 1.0 cores that have a reset input or output can be converted to OCP-IP 2.0 cores by renaming the Reset_n pin in the core's RTL conf file without touching the actual HDL source of the core. The new name depends on whether the reset is an input or output and whether the core is a master or slave.

In the very unlikely situation of an OCP-IP 1.0 core lacking a reset input or output, the conversion to OCP-IP 2.0 is achieved by the addition of a dummy reset input pin that is not used inside the core.

Debug and Test Interface

There are three debug and test interface extensions predefined for the OCP: scan, clock control, and IEEE 1149. The scan extension enables internal scan techniques, either in a pre-designed hard-core or end user inserted into a soft-core. Clock Control extensions assist in scan testing and debug when the IP core has at least one other clock domain that is not derived from Clk. The IEEE 1149 extension is for interfacing to cores that have an IEEE 1149.1 test access port built-in and accessible. This is primarily the case with cores, such as microprocessors, that were derived from standalone products.

These three extensions along with sideband signals (flags) can yield a highly debuggable and testable IP core and device.

Scan Control

The width and meaning of the Scanctrl field is user-defined. At a minimum this field carries a signal to specify when the device is in scan chain shifting mode. The signal can be used for the scan clock if scan-clock style flip-flops are being used. When this is a multi-bit field, another common signal to carry would be one specifying the scan mode. This signal can be used to put the IP core into any special test mode that is necessary before scanning and application of ATPG vectors can begin.

Clock Control

The clock control test extensions are included to ease the integration of IP cores into full or partial scan test environments and support of debug scan operations in designs that use clock sources other than Clk.

When an external clock source exists (for example, non-Clk derived clock), the ClkByp signal specifies a bypass of the external clock. In that case the TestClk signal usually becomes the clock source. The TestClk toggles in the correct sequence for applying ATPG vectors, stopping the internal clocks, and doing scan dumps as required by the user.

10 Timing Guidelines

To provide core timing information to system designers, characterize each core into one of the following timing categories:

- Level0 identifies the core interface as having been designed without adhering to any specific timing guidelines.
- Level1 timing represents conservative interface timing.
- Level2 represents high performance interface timing.

One category is not necessarily better than another. The timing categories are an indication of the timing characteristics of the core that allow core designers to communicate at a very high level about the interface timing of the core. Table 27 represents the inter-operability of two OCP interfaces.

Table 27 Core Interface Compatibility

	Level0	Level1	Level2
Level0	X	X	X
Level1	X	V	V
Level2	X	V	V*

X no guarantee

V guaranteed inter-operability with possible performance loss (extra latency)

V* high performance inter-operability but some minor changes may be required

The timing guidelines apply to dataflow and sideband signals only. There is no timing guideline for the scan and test related signals.

Timing numbers are specified as a percentage of the minimum supported clock-cycle (at maximum operating frequency). If a core is specified at 100MHz and the c2qtime is given as 30%, the actual c2qtime is 3ns.

Level0 Timing

Level0 timing indicates that the core developer has not followed any specific guideline in designing the core interface. There is no guarantee that the interface can operate with any other core interface. Inter-operability for the core will need to be determined by comparing timing specifications for two interfaces on a per-signal basis.

Level1 Timing

Level1 timing indicates that a core has been developed for minimum timing work during system integration. The core uses no more than 25% of the clock period for any of its signals, either at the input (setuptime) or at the output (outputtime). A core interface in this category must not use any of the combinational paths allowed in the OCP interface.

Since inputs and outputs each only use 25%, 50% of the cycle remains available. This means that a Level1 core can always connect to other Level1 and Level2 cores without requiring any additional modification.

Level2 Timing

Level2 timing indicates that a core interface has been developed for high performance timing. A Level2 compliant core provides or uses signals according to the timing values shown in Table 28. There are separate values for single-threaded and multi-threaded OCP interfaces. The number for each signal indicates the percentage of the minimum cycle time at which the signal is available, that is the outputtime at the output. Setuptime at the input is calculated by subtracting the number given from the minimum cycle time. For example, a time of 30% indicates that the outputtime is 30% and the setuptime is 70% of the minimum clock period.

In addition to meeting the timing indicated in Table 28, a Level2 compliant core must not use any combinational paths other than the preferred paths listed in Table 29.

There is no margin between outputtime and setuptime. When using Level2 cores, extra work may be required during the physical design phase of the chip to meet timing requirements for a given technology/library.

Table 28 Level2 Signal Timing

Signal	Single-threaded Interface %	Multi-threaded Interface %
Control, Status	25	25
ControlBusy, StatusBusy	10	10
ControlWr, StatusRd	25	25
Datahandshake Group (excluding MDataThreadID)	30	60
MDataThreadID	n/a	50
MRespAccept	50	75
MThreadBusy	10	10
MThreadID	n/a	50
Request Group (excluding MThreadID)	30	60
MReset_n, SReset_n	10	10
Response Group (excluding SThreadID)	30	60
SCmdAccept	50	75
SDataAccept	50	75
SDataThreadBusy	10	10
SError, SFlag, SInterrupt, MFlag, MError	40	40
SThreadBusy	10	10
SThreadID	n/a	50

Table 29 Allowed Combinational Paths for Level2 Timing

Core	From	To
Master	SThreadBusy	Request Group
	SThreadBusy SDataThreadBusy	Datahandshake Group
	Response Group	MRespAccept
Slave	MThreadBusy	Response Group
	Request Group	SCmdAccept and SDataAccept
	Datahandshake Group	SCmdAccept and SDataAccept

11 *Verification Guidelines*

This chapter describes checks (performed by the ocpcheck tool) to determine compliance with the OCP protocol and is intended to help in the development of verification suites and checkers in various verification environments. The checks within this chapter assume that the configuration being tested is valid. For more information about the ocpcheck tool, see the *Core Preparation Guide*.

Checks need to be applied with varying frequency; cycle-based checks on every cycle, phase-based checks across each phase of an OCP transfer. Transfer-level checks verify correct sequencing among grouped OCP phases. Transaction-level checks verify that protocol requirements for grouped transfers are not violated.

Make all OCP checks synchronous with respect to the rising edge of the OCP clock so that signal values are analyzed at the rising edge of the OCP clock. Check any complex hierarchy elements that span multiple cycles from data collected on the rising edge of the clock.

Checks of any sort should not be performed until either MReset_n or SReset_n becomes 0. Checks should also not occur if either signal is 0. The only checks that need not comply with this rule are the checks for X or Z on MReset_n and SReset_n. These checks are performed once either signal transitions to 0. A second exception is the check to see that MReset_n and SReset_n are held to 0 for sixteen cycles.

Signal Testing

Signal testing checks listed in Tables 30-33, verify that the values of the OCP signals are valid. Whenever a signal is valid at the rising edge of the OCP clock it should not have a value of X or Z.

Table 30 Signal Checks Every Cycle for X or Z

Signal	Enabling Parameter Expression
ControlBusy	controlbusy == 1
ControlWr	controlwr == 1
MCmd	-
MDataValid	datahandshake == 1
MError	merror == 1
MReset_n	mreset == 1
MThreadBusy	mthreadbusy == 1
SDataThreadBusy	sdatathreadbusy == 1
SError	serror == 1
SInterrupt	Interrupt == 1
SReset_n	sreset == 1
SResp	resp == 1
StatusBusy	statusbusy == 1
StatusRd	statusrd == 1
SThreadBusy	sthreadbusy == 1

Table 31 Signal Checks During Request Phases

Signal	Enabling Parameter Expression
MAddr	addr == 1
MAddrSpace	addrspace == 1
MAtomicLength	atomiclength == 1
MBurstLength	burstlength == 1
MBurstPrecise	burstprecise == 1
MBurstSeq	burstseq == 1
MBurstSingleReq	burstsinglereq == 1
MByteEn	byteen == 1
MConnID	connid == 1
MReqLast	reqlast == 1
MThreadID	threads > 1
SCmdAccept	cmdaccept == 1

Table 32 Signal Checks During Datahandshake Phase

Signal	Enabling Parameter Expression
MDataByteEn	mdatabyteen
MDataLast	datalast == 1
MDataThreadID	threads > 1
SDataAccept	dataaccept == 1

Table 33 Signal Checks During Response Phase

Signal	Enabling Parameter Expression
MRespAccept	respaccept == 1
SRespLast	resplast == 1
SThreadID	threads > 1

The signals listed below do not need to be checked for X or Z since they have no architecturally defined encodings.

Control	MData
MDataInfo	MFlag
MReqInfo	SData
SDataInfo	SFlag
SRespInfo	Status
Test	

Signal Retraction Testing

If a phase lasts more than one cycle checks need to be performed on some signals belonging to the corresponding group to ensure that they have not changed value from the starting cycle of the request phase.

Table 34 Signal Checks for Retraction During Request Phase

Signal	Enabling Parameter Expression
MAddr	addr == 1
MAddrSpace	addrspace == 1
MAtomicLength	atomiclength == 1
MBurstLength	burstlength == 1
MBurstPrecise	burstprecise == 1
MBurstSeq	burstseq == 1
MBurstSingleReq	burstsinglereq == 1
MByteEn	byteen == 1
MCmd	-
MConnID	connid == 1

Signal	Enabling Parameter Expression
MData	(mdata == 1) && (datahandshake == 0) and MCmd is a write type operation
MDataInfo	(mdatainfo == 1)&& (datahandshake == 0)
MReqInfo	reqinfo==1
MReqLast	reqlast == 1
MThreadID	threads > 1

Table 35 Signal Checks for Retraction During Datahandshake Phase

Signal	Enabling Parameter Expression
MData	(mdata == 1) && (datahandshake == 1)
MDataByteEn	mdatabyteen
MDataInfo	(mdatainfo == 1)&& (datahandshake == 1)
MDataLast	datalast == 1
MDataThreadID	threads > 1

Table 36 Signal Checks for Retraction During Response Phase

Signal	Enabling Parameter Expression
SData	(sdata == 1) and MCmd is a read type operation
SDataInfo	sdatainfo == 1
SResp	resp == 1
SRespInfo	respinfo==1
SRespLast	resplast == 1
SThreadID	threads > 1

Phase-Based Checks

Phase-based checks should be performed on each of the different phases. The checks should be independent of events at the transfer and transaction levels, and only examine an individual phase.

Request Phase Checks

The following checks must be performed during the request phase:

- For every cycle, verify that MCmd is not an illegal command. Illegal commands are those lacking an enabling parameter on the OCP interface. For example WRNP is not valid if `writenonpost_enable` and `writeresp_enable` are not enabled. Refer to Table 19 on page 47 for details on how to determine which commands are not enabled on an OCP connection.
- Any single request must have a MReqLast value of 1.

- When threads is greater than 1, check MThreadID to determine if the threads parameter is within the specified range.
- Check if a request is made to a specific thread that has SThreadBusy asserted. If sthreadbusy_exact is set to 1, an error should be logged.
- If force_aligned is enabled the byte enable pattern on MByteEn must represent an aligned subword that is a power-of-two bytes in size.
- MAddr must be OCP word aligned. The lowest order bits of MAddr may not be OCP word aligned as long as they are zero.
- MBurstSeq can not use the reserved value 0x7.

Datahandshake Phase

The following checks must be performed when datahandshake is enabled on the OCP interface:

- Every datahandshake phase must have a matching request phase.
- Check that MDataThreadID is within a valid range, that is, MDataThreadID < threads.
- Check if a datahandshake is made to a thread with SDataThreadBusy asserted. If sdatathreadbusy_exact is set to 1, an error should be logged.
- If force_aligned is enabled the byte enable pattern MDadaByteEn must represent an aligned subword that is a power-of-two bytes in size.

Response Phase

Perform the following checks for all response phases:

- Every response phase must have a corresponding request phase.
- Check that SThreadID is within a valid range, that is, SThreadID < threads.
- Check if a response is sent to a thread with MThreadBusy asserted. If mthreadbusy_exact is set to 1, an error should be logged.

Transfer-Based Checks

Every transfer should be checked for the following:

- A datahandshake phase cannot begin before the associated request phase begins, but can begin in the same Clk cycle.
- A datahandshake phase cannot end before the associated request phase ends, but can end in the same Clk cycle.
- A response phase cannot begin before the associated request phase begins, but can begin in the same Clk cycle.

- A response phase cannot end before the associated request phase ends, but can end in the same Clk cycle.
- If there is a datahandshake phase and a response phase, the response phase cannot begin before the associated datahandshake phase begins, but can begin in the same Clk cycle.
- If there is a datahandshake phase and a response phase, the response phase cannot end before the associated datahandshake phase ends, but can end in the same Clk cycle.
- Check that response is allowed for the given MCmd.
- For a single transfer requiring a datahandshake phase, if MDataLast is configured it must be asserted.
- For a single transfer requiring a response phase, if SRespLast is configured it must be asserted.

The ordering rules also apply to transfers within a single request/multiple data (SRMD) burst. For write SRMD burst transfers all the datahandshake phases of a burst match to one common request phase. If `writeresp_enable` is set to 1, all of the datahandshake phases of a burst also match to one common response phase. For read SRMD burst transfers all the response phases of a burst match to one common request phase.

Check for incomplete transfers at the end of simulation. If a transfer has not started one of its phases or completed one of its phases, report the incomplete transfers that are detected.

For multi-threaded OCP interfaces with `datahandshake` set to 1 and `sdatath-readbusy` set to 0, check that the order of the datahandshake phases follows the order of the requests phases across all threads.

Transaction-Based Checks

Check completed transfers to determine if there are protocol violations across multiple transfers. Test all burst and RDEX transactions to ensure that no transfers are missing and that each transfer is complete. In the event an incomplete transaction is detected at the end of simulation, it may be adequate to report only the starting times of the detected incomplete transactions.

Burst Checks

When the transaction consists of one or more transfers with the first transfer having an MBurstLength >1, perform the following checks:

- Check that RDEX, RDL, and WRC are not part of a burst. It is sufficient to just check the starting transfer of the burst.

- MCmd, MConnID, MAddrSpace, MBurstPrecise, MBurstSingleReq, MBurstSeq, MAtomicLength, MReqInfo and SRespInfo must remain constant across all transfers of the burst. This is true for both precise and imprecise bursts.
- When MBurstSingleReq is set to 0, for the last transfer of the burst, MReqLast must be 1. All other burst transfers must have a MReqLast of 0. When MBurstSingleReq is set to 1, MReqLast must be 1.
- MBurstLength must remain constant across all transfers of a precise burst.
- MDataLast should only be 1 for the last transfer of the burst. All other burst transfers must have a MDataLast value of 0.
- For the last transfer of a burst, SRespLast must be 1. All other burst transfers must have a SRespLast value of 0.
- If `burstseq_dflt1_enable` is 0 the master must not issue DFLT1 bursts. This can be determined by checking the starting transfer of the burst.
- If `burstseq_dflt2_enable` is 0 the master must not issue DFLT2 bursts. This can be determined by checking the starting transfer of the burst.
- If `burstseq_incr_enable` is 0 the master must not issue INCR bursts. This can be determined by checking the starting transfer of the burst.
- If `burstseq_strm_enable` is 0 the master must not issue STRM bursts. This can be determined by checking the starting transfer of the burst.
- If `burstseq_unkn_enable` is 0 the master must not issue UNKN bursts. This can be determined by checking the starting transfer of the burst.
- If `burstseq_wrap_enable` is 0 the master must not issue WRAP bursts. This can be determined by checking the starting transfer of the burst.
- If `burstseq_xor_enable` is 0 the master must not issue XOR bursts. This can be determined by checking the starting transfer of the burst.
- WRAP and XOR burst sequences must only occur with precise bursts.
- WRAP and XOR must always have a power of two burst length. Checking MBurstLength of the first transfer is sufficient for this check.
- Verify the MAddr sequence is correct for XOR bursts.
- For incrementing bursts, when `burst_aligned` is enabled, the total burst size must be a power of two.
- When `burst_aligned` is enabled, incrementing bursts must have their starting address aligned with the total burst size.
- When `burst_aligned` is enabled, incrementing bursts must be issued as precise bursts.
- Each transfer of an incrementing burst must increment MAddr by the OCP word size.

- For incrementing bursts MAddr must never wrap around the address space of the OCP interface.
- Streaming bursts must have the same MAddr across all transfers of the burst and MByteEn must be constant
- Verify that the address sequence is correct for WRAP bursts.
- If reqdata_together is set to 1, the request and data must arrive together.
- For precise bursts the transfer count must match MBurstLength.

Read Exclusive Transaction Check

Perform the following checks on RDEX transactions:

- On the completion of a ReadEx transfer, check that the next transfer on the same thread is a WR or WRNP to the same address (MAddr and MAddrSpace), and byte enable pattern (MByteEn) as the ReadEx transaction.
- Check that the WR/WRNP following the RDEX has a MBurstLength of 1

Sideband Checks

The checks in this section apply to the sideband signals.

Reset Checks

The following checks must be performed on the reset signals:

- If MReset_n is asserted, it must remain asserted for at least 16 cycles.
- If SReset_n is asserted, it must remain asserted for at least 16 cycles.

Control Checks

The following checks must be performed on the control signals:

- Control must be held steady the first cycle after reset is deasserted.
- The ControlWr signal cannot be asserted in the cycle following a reset.
- The Control signal can not change more than once every other cycle.
- The ControlWr signal must be asserted when the Control signal changes.
- The Control signal may not change if the ControlBusy signal is asserted.
- The ControlWr signal can not remain asserted for two consecutive cycles.
- The ControlWr signal can not be asserted if ControlBusy is active.

- ControlBusy can only be asserted in a cycle after ControlWr is asserted or reset transitions to inactive.

Status Checks

The following checks must be performed on the status signals:

- The StatusRd signal is active for no more than one clock cycle.
- The StatusRd signal is not asserted while StatusBusy is asserted.

12 *Core Performance*

To make it easier for the system integrator to choose cores and architect the system, an IP core provider should document a core's performance characteristics. This chapter supplies a template for a core performance report on page 180, and directions on how to fill out the template.

Report Instructions

To document the core, you will need to provide the following information:

1. Core name. Identify the core by the name you assigned.
2. Core ID. Specify the precise identification of the core inside the system-on-chip. The information consists of the vendor code, core code, and revision code.
3. Core is/is not process dependent. Specify whether the core is process-dependent or not. This is important for the frequency, area, and power estimates that follow.

   If multiple processes are supported, name them here and specify corresponding frequency/area/power numbers separately for each core if they are known.
4. Frequency range for this core. Specify the frequency range that the core can run at. If there are conditions attached, state them clearly.
5. Area. Specify the area that the core occupies. State how the number was derived and be precise about the units used.
6. Power estimate. Specify an estimate of the power that the core consumes. This naturally depends on many factors, including the operations being processed by the core. State all those conditions clearly, and if possible, supply a file of vectors that was used to stimulate the core when the power estimate was made.

7. Special reset requirements. If the core needs MReset_n/SReset_n asserted for more than the default (16 OCP clock cycles) list the requirement.

8. Number of interfaces.

9. Interface information. For each OCP interface that the core provides, list the name and type.

   The remaining sections focus on the characteristics and performance of these OCP interfaces.

   - For master OCP interfaces:

     a. Issue rate (per OCP cycle for sequences of reads, writes, and interleaved reads/writes). State the maximum issue rate. Specify issue rates for sequences of reads, writes, and interleaved reads and writes.

     b. Maximum number of operations outstanding (pipelining support). State the number of outstanding operations that the core can support; is there support for pipelining.

     c. If the core has burst support, state how it makes use of bursts, and how the use of bursts affects the issue rates.

     d. High level flow-control. If the core makes use of high-level flow control, such as full/empty bits, state what these mechanisms are and how they affect performance.

     e. If multiple threads are present, explain the use of threads.

     f. Connection ID support. Explain the use and meaning of connection information.

     g. Use of side-band signals. For each sideband signal (such as SInterrupt, MFlag) explain the use of the signal.

     h. If the OCP interface has any implementation restrictions, they need to be clearly documented.

   - For slave OCP interfaces:

     a. Unloaded latency for each operation (in OCP cycles). Describe the unloaded latency of each type of operation.

     b. Throughput of operations (per OCP cycle for sequences of reads, writes, and interleaved reads/writes). State the maximum throughput of the operations for sequences of reads, writes, and interleaved reads and writes.

     c. Maximum number of operations outstanding (pipelining support). State the number of outstanding operations that the core can support, i.e. is there support for pipelining.

     d. Burst support and effect on latency and throughput numbers. If the core has burst support, state how it makes use of bursts, and how the use of bursts affects the latency and throughput numbers stated above.

e. High level flow-control. If the core makes use of high-level flow control, such as full/empty bits, state what these mechanisms are and how they affect performance.

f. If multiple threads are present, explain the use of threads.

g. Connection ID support. Explain the use and meaning of connection information.

h. Use of side-band signals. For each sideband signal (such as SInterrupt, MFlag) explain the use of the signal.

i. If the OCP interface has any implementation restrictions, they need to be clearly documented.

- For every non-OCP interface, you will need to provide all of the same information as for OCP interfaces wherever it is applicable.

Sample Report

1. Core name	flashctrl
2. Core identity Vendor code Core code Revision code	 0x50c5 0x002 0x1
3. Core is/is not process dependent	Not
4. Frequency range for this core	≤100Mhz with NECCBC9-VX library
5. Area	4400 gates 2Input NAND equivalent gates
6. Power estimate	not available
7. Special reset requirements	
8. Number of interfaces	2
9. Interface information: Name Type	 ip slave
For master OCP interfaces:	
a. Issue rate (per OCP cycle for sequences of reads, writes, and interleaved reads/writes)	
b. Maximum number of operations outstanding (pipelining support)	
c. Effect of burst support on issue rates	
d. High level flow-control	
e. Use of threads (if any)	
f. Use of connection information	
g. Use of side-band signals	
h. Implementation restrictions	

For slave OCP interfaces:	
a. Unloaded latency for each operation (in OCP cycles)	Register read or write: 1 cycle. The flash read takes SBFL_TAA (read access time). Can be changed by writing corresponding register field of emem configuration register. The flash write operation takes about 2000 cycles since it has to go through the sequence of operations - writing command register, reading the status register twice.
b. Throughput of operations (per OCP cycle for sequences of reads, writes, and interleaved reads/ writes)	No overlap of operations therefore reciprocal of latency.
c. Maximum number of operations outstanding (pipelining support)	No pipelining support.
d. Effect of burst support on latency and throughput numbers	No burst support.
e. High level flow-control	No high-level flow-control support.
f. Use of threads (if any)	No thread support.
g. Use of connection information	No connection information support.
h. Use of side-band signals	Reset_n, Control, SError. Control is used to provide additional write protection to critical blocks of flash memory. SError is used when an illegal width of write is performed. Only 16 bit writes are allowed to flash memory.
i. Implementation restrictions	
For every non-OCP interface Provide all of the same information as for OCP interfaces wherever it is applicable.	Hitachi flash card HN29WT800 Only 1 flash ROM part is supported, therefore the CE_N is hardwired on the board. The ready signal RDY_N, is not used since not all parts support it. For the BYTE_N signal, only 16-bit word transfers are supported

Performance Report Template

Use the following template to document a core.

1. Core name	
2. Core identity Vendor code Core code Revision code	
3. Core is/is not process dependent	
4. Frequency range for this core	
5. Area	
6. Power estimate	
7. Special reset requirements	
8. Number of interfaces	
9. Interface information: Name Type	
For master OCP interfaces:	
a. Issue rate (per OCP cycle for sequences of reads, writes, and interleaved reads/writes)	
b. Maximum number of operations outstanding (pipelining support)	
c. Effect of burst support on latency and throughput numbers	
d. High level flow-control	
e. Use of threads (if any)	
f. Use of connection information	
g. Use of side-band signals	
h. Implementation restrictions	

For slave OCP interfaces:	
a. Unloaded latency for each operation (in OCP cycles)	
b. Throughput of operations (per OCP cycle for sequences of reads, writes, and interleaved reads/ writes)	
c. Maximum number of operations outstanding (pipelining support)	
d. Effect of burst support on latency and throughput numbers	
e. High level flow-control	
f. Use of threads (if any)	
g. Use of connection information	
h. Use of side-band signals	
i. Implementation restrictions	
For every non-OCP interface Provide all of the same information as for OCP interfaces wherever it is applicable.	

Index

C

J

L

M

R

S

U

V

W

X

OCP-IP Association
5440 SW Westgate Drive, Suite 217
Portland, OR 97221
Ph: 503-291-2560
Fax: 503-297-1090
admin@ocpip.org
www.ocpip.org

Part Number: 161-000125-0002

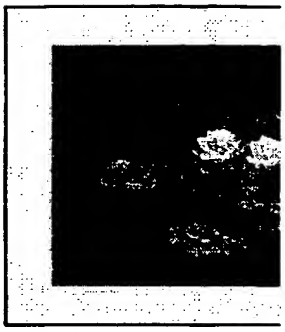
http://java.sun.com/developer/technicalArticles/Threads/applet/ May 20, 2005

Article
Creating a Threaded Slide Show Applet

Articles Index | Threads Index

By Monica Pawlan
March 16, 2001

A thread is a path of execution through a program. Single threaded programs have one path of execution, and multi-threaded programs have two or more paths of execution. Single threaded programs can perform only one task at a time, and have to finish each task in sequence before they can start another. For most programs, one thread of execution is all you need, but sometimes it makes sense to use multiple threads in a program to accomplish multiple simultaneous tasks.



For example, a multi-threaded math program can let the user set paremeters for a new calculation while a previous calculation computes. Or a multi-threaded word processor can let the user open a new document while a large file saves or spools to the printer. And sometimes, as you will see in this article, a multi-threaded approach is necessary when you need a method to execute in a continuous loop in one thread and another thread to do something else like paint the display between loop iterations.

This article describes multi-threaded programming by presenting an example slide show applet. The images for the slide show applet are thumbnails of paintings by Claude Monet downloaded from a public web site. If Claude Monet were alive today, the thumbnails and slide show applet might very well be available from his own web site so prospective buyers can view his latest works and make electronic purchases.

All single or multi-thread programs execute in their own thread created and started by the underlying Java VM[1]. The Java VM also creates a threads to help manage single or multi-threaded program execution. For example, the Java VM starts the `Finalizer` thread to execute an method before the object is garbage collected, and starts the `AWT-EventQueue-0` thread to call an object's event handling methods such a `actionPerformed` and `windowClosing`. Because the Java VM spawns threads to handle the execution of a program, you might say that is a multi-threaded program.

Normally, you do not have to concern yourself with threads created by the Java VM to manage your program execution. As a developer, you is to determine whether your program works best single threaded or multi-threaded, and then design and implement it accordingly. In making should take into account that spawning additional threads carries overhead by consuming extra memory and processor resources. But some threaded approach is the best way to go so the user does not have to wait too long for a task to complete before starting another, or as in the show applet, for the program to work at all.

Running the Slide Show Applet

Slide One

The slide show applet downloads three thumbnail paintings by Claude Monet and draws them one-by-one on the applet's panel. The thumbnail paintings are stored on a Web site and the slide show applet accesses them by their URL.

To run the slide show applet, get the SlideShow.java class file and the java.policy security policy file. Compile the `SlideShow.java` class file and create a `slides.html` file that looks like this:

```
<HTML>
<BODY>
  <APPLET CODE=SlideShow.class
    WIDTH=150
    HEIGHT=150>
  </APPLET>
</BODY>
</HTML>
```

Use the applet viewer tool to run the applet as follows:

```
appletviewer -J-Djava.security.policy=
java.policy slides.html
```

Note: To run an applet written with Java 2 APIs in a browser, the browser must be enabled for the Java 2 platform. If your browser is not enabled for the Java 2 Platform, you ha use the the applet viewer tool to run the applet or install Java Plug-In. Java Plug-In lets you run applets on Web pages under the 1.2 version of the Java VM instead of the Web browser's default VM. For information on getting setup, see Chapter 9 "Deploying the Auction Application" in Advanced Programming for the Java 2 Platform.

How the Threaded Applet Works

Slide Two

The slide show applet executes in its own thread and spawns a second thread to retrieve images from an image array, retrieve captions from a text array, and call the `repaint` method to initiate drawing the image and caption text. The actual drawing of the image and text is handled by an event thread started by the Java VM.

Every thread has a name, so you can tell which thread is doing what by retrieving the current thread and getting its name with the `getName` method in the `Thread` class. When you create program, you can supply an optional name, and if you do not supply a name, the Java VM provides a default name that follows the following naming convention: `Thread-0`, `Thread-1`, an created by the Java VM have names too, as you will see in the sample run below.

Slide Show Sample Run

The slide show applet uses the `getName` and `System.out.println` methods to show which thread is executing. Below is a sample slide show applet run that goes through all the applet's life cycle methods.

As you can see in the sample run, the `init`, `start`, `stop`, and `destroy` life cycle methods execute in `thread applet-SlideShow.class`. The `run` method executes in the `Timer` thread, and gets called after the `Timer` thread is created and started in the applet's `start` method. The `Timer` thread retrieves images and captions from their arrays and calls the `repaint` method to initiate drawing to the applet's panel. The applet's `update` and `paint` methods do the actual drawing and execute in the `AWT-EventQueue-0` thread because updating and painting are panel events.

```
init() in thread applet-SlideShow.class
start() in thread applet-SlideShow.class
run() in Timer
update() in AWT-EventQueue-0
paint() in AWT-EventQueue-0
stop() in thread applet-SlideShow.class
start() in thread applet-SlideShow.class
destroy() in thread applet-SlideShow.class
```

New threads are spawned from threads that are already running. The Java VM spawns `thread applet-SlideShow.class` and `AWT-EventQueue-0` to handle applet execution and panel events, and `thread applet-SlideShow.class` spawns the `Timer` thread. The spawning of the `Timer` thread is what makes this program multi-threaded.

Why Use The Timer Thread?

On the surface, it might seem like the applet has enough threads without creating the `Timer` thread. After all, the applet's `start` method running in `thread applet-SlideShow.class` can retrieve the images and text and call `repaint`, and the `update` and `paint` methods running in the `AWT-EventQueue-0` thread can do the rendering. Right? Well, not really.

The browser's VM calls into `init`, `start`, `stop`, and `destroy` and expects them to return after doing whatever it is they have to do. If you change the applet so `init` or `start` or a method called by `init` or `start` retrieves the images and text, the method will execute forever, never return, and put the applet in a deadlocked state. When that happens the only way to stop the applet is to issue a kill, interrupt, or break signal from the command line. The `Timer` thread is needed so `init` or `start` can return and let the applet paint.

Threaded Applet Anatomy

The SlideShow applet implements the following methods:

Life Cycle methods:
- `init`
- `start`
- `stop`
- `destroy`

Thread-related methods:
- `startThread`
- `run`
- `doPaused`
- `waitWhilePaused`

Drawing-related methods:
- `paint`
- `update`

Note: This section describes the slide show applet in terms of threads and how the applet's panel is updated with new images and captions. If you need to brush up on applet basics, see the applets lesson in Essentials of the Java Programming Language: A Hands-On Guide.

Initializing the Applet

The `init` method creates the slide show display and arrays to hold the images and caption text. The images are downloaded from a public Web site, so a URL class provides the URL for the the images. This method calls the `startThread` method, which creates the `Timer` thread and leaves it in a paused state waiting to be notified to begin execution.

```
public void init() {
    images = new Image[11];
    text = new String[11];
    setLayout(new BorderLayout());
    setFont(new Font("SansSerif", Font.BOLD, 24));

    Label name = new Label("by Claude Monet");
    name.setAlignment(Label.CENTER);
    add("South", name);

    URL tmp = null;
    try {
      tmp = new URL(
"http://www.javasoft.com:80/"+
"developer/technicalArticles/Threads/applet/");
    } catch (java.net.MalformedURLException ex) {
      System.out.println("Bad URL");
      return;
    }
```

```
   images[ [illegible] ] = getImage( [illegible] "[illegible]gherm.jpg");
   images[ [illegible] ] = getImage( [illegible] "[illegible]kaat.jpg");
   images[ [illegible] ] = getImage( [illegible] "Leyden.jpg");
   text[ [illegible] ] = "[illegible]";
   text[ [illegible] ] = "[illegible] Zuidkaat";
   text[ [illegible] ] = "[illegible] Windmill near Leyden";
   printThreadName( [illegible] );

   startThread();
 }
```

The printThreadName([illegible]) implementations gets the name of the current thread and prints that name to the command line. The sample run above shows that at this point in t execution the current thread is the applet thread.

```
init is thread applet-SlideShow.class
```

Starting the Thread

The startThread method is called once when the applet is first initialized to create timerThread, initialize variables, and implement [illegible] behavior. The thread remains in exis stopped in the applet's destroy method.

```
private void startThread() {
  paused = true;
  noStopRequested = true;

  // Use this inner class to hide
  // the public run method
  Runnable r = new Runnable() {
    public void run() {
      runWork();
    }
  };
  timerThread = new Thread(r, "Timer");
  timerThread.start();
}
```

The line timerThread = new Thread(this, "Timer") means create a thread named Timer and use this object as its target runnable. A target runnable is an object of type Runn method implementation. So what you are really saying is when the Timer thread starts, execute the run method implementation in this (the SlideShow) object.

The call to timerThread.start() on the next line enlivens the Timer thread by calling the run method in its target runnable. A thread remains alive until the run method returns.

The SlideShow class uses an inner class to create the target runnable and hide the public run method. An inner class is a class nested or defined inside another class, and hiding the pul implementation in an inner class like this prevents external code from erroneously calling it. Because the run method is public, using an inner class is safer than having the SlideShow cla Runnable interface and providing the run method implementation as part of the SlideShow class like this:

```
public class SlideShow extends Applet
                       implements Runnable {
...
public void run()
  runWork();
```

An interface defines a set of methods but does not implement their behavior. Instead, you provide the interface method implementations for the class that implements the interface. The Ru provides the run method as a common protocol for a class instance to execute a separate thread of code. Your implementation for the run method defines what the separate thread of exe In this example, the run method implementation calls the runWork method described in Running The Timer Thread below.

Using the Runnable interface is one of two approaches available in the Java programming language for writing multi-threaded program. The other approach is to extend the Thread class the Runnable interface. However, because the Java programming language does not let you extend more than one class, extending the Thread class has its limitations when you need to extend the Applet class to write an applet or extend the Frame class to write an application with a graphical user interface. For this reason, using the Runnable interface is the more com

The wait-notify mechanism is used to keep the timerThread in the paused state until the applet's start method is called. The wait-notify mechanism allows one thread to wait for a notifi thread that it can proceed. The paused variable is set to true in the startThread method and checked by the Timer thread when the run method is called. Calls to the applet's start toggle the paused variable between false and true, respectively.

Note that the noStopRequested instance variable set to false in the startThread method is declared volatile. The volatile modifier requests the Java VM to always access the variable so the its most current value is always read. Making noStopRequested volatile is necessary so other threads see the changed value. In this example thread applet-SlideSh changes the value in its startThread and stopThread methods and the Timer thread reads the value in its run method.

> **Note:** It is important to understand when to use the volatile modifier, and there are some rule you can follow. To quote a section from Java Thread Programming: The Autho Solution by Paul Hyde:
>
> If
>
> - two or more threads access a member variable, and
> - one or more threads might change that variable's value, and
> - all of the threads do not use synchronization (methods or blocks) to read and/or write the value.

then

that member variable must be declared volatile to ensure all threads see the changed value.

See Pausing the Applet for information on using synchronization.

Running The Timer Thread

The runWork method moves into a while loop if no stop has been requested and stays in the loop until the applet's stopThread method is called where the noStopRequested variable is set to false. Inside the loop the waitWhilePaused method is called to keep the applet in a paused state if the paused instance variable has a value of true.

When the applet first starts, its paused variable is initialized to true and changed to false when the start method is called. It is also set to true when the applet's stop method is called. So the Timer thread sits in the while loop waiting for the applet's start and stop methods to set the values for the paused variable.

When paused is set to false, the Timer thread retrieves images and captions from their arrays and calls the repaint method to update and paint the applet's panel. Between calls to the repaint method and retrieving the next image and caption, the thread sleeps for 3 seconds so each image in the slide show can stay on the display for viewing.

The sleep method throws InterruptedException, which is raised when timerThread.interrupt is called from the stopThread method. The stopThread method is called by the applet's destroy method when the applet exits. The interrupt causes the Timer thread to move out of sleep mode and jump to the catch block where the interrupt is reasserted, a message is printed, and the thread dies.

Note that the curFrame instance variable is declared volatile because it is updated by the runWork method in the Timer thread and read by the update method in thread applet-SlideShow.class.

```
// Note this method is private
  private void runWork() {
    printThreadName("run is ");
    curFrame = 0;

    try {
      while ( noStopRequested ) {
        waitWhilePaused();

        curFrame = ( curFrame + 1 ) %
                        images.length;
        repaint();

        Thread.sleep(3000);
      }
    catch ( InterruptedException x ) {
     // reassert interrupt
     Thread currentThread().interrupt();
     System.out.println(
            "interrupt and return from run");
    }
```

The call to printThreadName at the beginning gets the name of the current thread and prints it to the command line. The sample run above shows that at this point in the applet's execution the current thread is the Timer thread:

```
run as Timer
```

Pausing the Applet

The mechanism for pausing the applet consists of the pauseLock object, paused instance variable, setPaused method, and waitWhilePaused method. The paused instance variable is not declared volatile because all threads that update or read it use synchronization.

Synchronization lets one thread safely change values that another thread reads. The setPaused method executed by thread applet-SlideShow.class toggles the paused variable between true and false according to whether the applet is in a stopped, started, or initialized but not yet started state. The waitWhilePaused method executed by the Timer thread reads the value and keeps the applet in a paused state when the paused value is set to true and in an un-paused state when the paused value changes to false. Because both these methods use synchronization, the waitWhilePaused method cannot read the value of the paused variable until the setPaused method finishes setting it and vice versa.

When the Timer thread calls waitWhilePaused, it synchronizes on pauseLock and reads the value of paused. If the value is true, pauseLock.wait is called and the Timer thread goes to sleep. What is happening is the Timer thread is waiting on the pauseLock object to prevent the waitWhilePaused method from returning. If the value is false, the while loop in the Timer thread continues to execute.

If the Timer thread is put to sleep, it stays asleep until it is either interrupted or notified by the setPaused method that paused is now false. When thread applet-SlideShow.class calls setPaused, it synchronizes on pauseLock, changes the value of paused, notifies all waiting threads the value has changed, exits the synchronized block, and returns. When the Timer thread receives the notification, it wakes up, exits its synchronized block, and returns from the waitWhilePaused method.

```
  private void setPaused(boolean newPauseState) {
    synchronized ( pauseLock ) {
      if ( paused != newPauseState ) {
        paused = newPauseState;
        pauseLock.notify();
      }
    }
```

```
  }

  private void waitWhilePaused()
                          throws InterruptedException {
    synchronized ( pauseLock ) {
      while ( paused ) {
        pauseLock.wait();
      }
    }
```

Object-Level Locks

When code synchronizes on an object, it acquires an object-level lock on that object. This means no other thread can call methods on that object until the lock is released.

The `wait` methods of the `Object` class require that an object-level lock be held before they are called. Very soon after the thread gets inside the native code of the `wait` method, the Java VM releases the lock that thread was holding. Once the waiting thread is notified, times out, or is interrupted, it competes with other threads to reacquire the object-level lock. Once the thread has exclusive access to the object-level lock, it returns from the `wait` method and continues inside the synchronized block.

A thread must hold an object-level lock on the object before it can call any of the `wait`, `notify`, or `notifyAll` methods on that object. Once a thread gets inside `wait`, the Java VM temporarily releases the lock to ensure that the signaling is solid and not open to any race conditions. And if the waiting thread did not release the lock, another thread would not be able to call `notify` or `notifyAll` because it would not be able to get the lock.

Updating and Painting the Display

The `repaint` method called from within the `run` method repaints the applet's panel and calls the applet's `paint` method to completely redraw the applet's panel.

```
  public void paint(Graphics g) {
    Thread t = Thread.currentThread();
    String paint = t.getName();
    System.out.println("paint is " + paint);
  }

  public void update(Graphics g) {
    g.drawImage(images[curFrame], 0, 0, this);
    captions.setText(texts[curFrame]);
    Thread t = Thread.currentThread();
    String update = t.getName();
    System.out.println("update is " + update);
  }
```

The call to `printThreadName` at the beginning gets the name of the current thread and prints it to the command line. The sample run above shows that at this point in the applet's execution the current thread is the AWT-EventQueue-0 thread:

```
update is AWT-EventQueue-0
paint is AWT-EventQueue-0
```

Stopping the Applet

The applet's `stop` method is called whenever you minimize applet viewer, or if the applet is running in a Web page, whenever you go to another Web page. At this point the `setPaused` method is called to put the applet in a paused state.

```
  public void stop() {
    setPaused(true);
    printThreadName("stop is ");
  }
```

The last line of the `stop` method calls `printThreadName` to get the name of the current thread and print it to the command line. The sample run above shows that at this point in the applet's execution the current thread is the the applet thread:

```
stop is thread applet-SlideShow.class
```

Exiting the Applet

When you exit applet viewer or close the browser, the applet's `destroy` method is called to stop the thread and perform cleanup. The image and text arrays are set to `null` to make them ready for garbage collection.

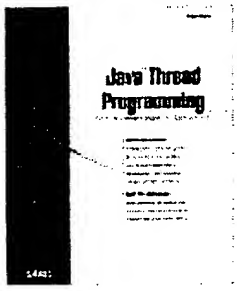
The call to `images[i].flush` flushes all resources used by the `Image` array, including any pixel data being cached for rendering to the screen and any system resources used to store data or pixels for the images. The images are reset to a state similar to when they were first created so that if they are rendered again, the image data has to be recreated or fetched from its source.

```
  public void destroy() {
    stopThread();
    for (int i = 0; i < images.length; i++) {
      images[i].flush();
      images[i] = null;
      texts[i] = null;
    }
    images = null;
    texts = null;
    System.out.println("destroy is ");
  }
```

The last line of the [illegible] method calls the [illegible] method to get the name of the current thread and print the name to the command line. The sample run above shows that at this point in the applet's execution the current thread is the the applet thread:

```
[illegible] applet.SlideShow.class
```

Conclusion and Further Reading



While it is relatively easy to get started with multi-threaded programming in the Java programming language, mastering the use of multiple threads and establishing communications among them takes time, practice, and patience. An examples-based approach is one of the best ways to learn because you reinforce what you read by putting the concepts into practice.

An excellent examples-based reference is Java Thread Programming: The Authoritative Solution by Paul Hyde. This is an accessible and complete treatment that inexperienced and experienced developers alike should find worthwhile.

The Doing Two or More Tasks at Once lesson in *The Java Tutorial* provides excellent introductory coverage as well.

Also see Concurrent Programming in Java by Doug Lea and visit his web site.

Monica Pawlan, a staff writer for the Java Developer Connection (JDC), is author of *Essentials of the Java Programming Language: A Hands-On Guide* (Addison-Wesley, 2000), and co-author of *Advanced Programming for the Java 2 Platform* (Addison-Wesley, 2000).

Have a question about programming? Use Java Online Support.

[1] As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.